Using Middle-Out Reasoning to Guide Inductive Theorem Proving

Jane Thurmann Hesketh

Ph.D.

University of Edinburgh

1991

I DECLARE THAT THIS THESIS HAS BEEN COMPOSED BY MYSELF AND THAT THE WORK DESCRIBED IN IT IS MY OWN:

(Jane Hesketh)

Acknowledgements

The author was supported by use of equipment and its maintenance provided by the Science and Engineering Research Council under grants GR/F/71799, GR/E/44598, GR/D/44874, and by the Alvey initiative under grant GR/D/44270. Further equipment maintenance was provided by Esprit BRA actions P3012 and P3245. Edinburgh University remitted the author's tuition fees. All that support is gratefully acknowledged.

Many people contributed to the work in very useful and practical ways. Special thanks are due to:

- My supervisors, Alan Bundy and Alan Smaill, who have been patient, encouraging, conscientious, constructive and constant sources of advice and comment at all levels;

- Ina Kraan and Andrew Ireland, who gave me helpful feedback on my thesis drafts;

- Helen Pain, for helping me out with all the practicalities towards the end;

- All the present and past members of the DReaM group, including those on the non-mathematical grants, who have made such a good working and research environment;

- My family and my friends, for patiently awaiting the end of my social disappearance;

- Phil Odor, for unfailing practical and friendly support, taking care of me at the stressful times, many provocative, challenging discussions, and for believing in me.

Abstract

Techniques derived from proof theory for logic alone have been insufficient as a basis for efficient, elegant automatic theorem proving. They concentrate on syntax, neglecting both strategy for particular domains and classes of problem, and guidance from modelling human mathematicians.

A novel technique suggested by Bundy, developing ideas from Ernst & Newell, is to reason "middle-out". Often, the overall structure of a proof may be known, but its details must be fleshed out according to the individual theorem. Conventional search might use heuristic guidance to backtrack over all possibilities. Middle-out reasoning uses variables as place-holders for parameters still to be chosen. These place-holders become instantiated by the requirements of the subsequent proof. Decisions which would multiply the search space are postponed until more information is available.

This is an exciting development in search control, making extensive use of strategic guidance and harnessing tools from human reasoning. This thesis reports research on its use for the synthesis of tail recursive functions from corresponding naive functions and for proofs requiring generalisation. It enables the development of a unified framework for generalisation.

An existing proof planning and development system based on Martin-Löf Type Theory, Oyster/CLAM, is used as a vehicle, augmented by higher order unification.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis reports research on middle-out reasoning (MOR), an attempt to replicate some of the mathematician's ability to speculate about the approximate nature of a proof, and then flesh out the details as required by the particular problem. Such speculation is difficult to harness, making considerable demands on the mathematician's experience and understanding of proof structure.

I have studied MOR in the context of two classes of problem:

- Synthesis of tail-recursive programs from naïvely recursive specifications.
- Generalisation of inductive theorems.

Each of these was rewarding in itself. Proof structures corresponding to tail-recursive programs have only been characterised theoretically. I developed and implemented an automatic system to create such proofs in a programs-as-proofs context. Generalisation has been handled by a number of specialised techniques. MOR made it possible to discover, represent and prove these generalisations within a unified framework. They all contributed to a common purpose, and followed a similar pattern, which MOR's flexibility could describe.

Developing automated MOR solutions for these problem areas also enabled me to form conclusions about MOR: its contribution to search control, to the problem areas I explored, and to a limited extent, what claim it has to psychological validity.

Some motivation for my choices of problem will be provided later in this chapter. Before that, I will describe MOR, expand a little on the subject of speculation in mathematics and then give a very brief overview of the topic of automated mathematical reasoning, to explain broadly how MOR relates to other work in the field as a whole. To end, I will give some implementation decisions and outline the structure of the rest of the thesis.

1.1 Middle-Out Reasoning

In recent years, *meta-level reasoning* techniques have had considerable success in a number of fields of automated reasoning. These techniques recognise a difference between on the one hand, a problem and the language which describes it (*the object level*), and on the other hand, the problem of reasoning *about* the object-level problem and the language which describes that (*the meta-level*). This explicit separation focuses attention on the control of reasoning, as opposed to the manipulation of symbols at the object-level. The meta-level will normally use representations of terms in the object level language in the course of reasoning.

The MOR technique, as suggested by Bundy is a development of Ernst & Newell's version for GPS [Ernst & Newell 69]. It extends meta-level reasoning, by allowing the representations of the object level to contain meta-variables, which represent object-level entities which may, if necessary, be higher-order. The entities the meta-variables stand for must be legitimate within the object level language. The significance of such meta-variables is that they can be used for unknown terms or formulae, postponing decisions about their identity until more information is available later, whereupon they become instantiated. This should be contrasted with the usual approach of heuristic selection from amongst known possible values at each stage.

There are a number of phases in MOR.

1. deciding that speculation should be attempted;

2. forming the speculation, using meta-variables;

3. exploring its consequences and finding values for the variables;

4. deciding whether it has succeeded, and possibly selecting from alternative solutions.

It is convenient to think of the first two of these as the *speculation*, and of the last two as resolving its identity. Unfortunately, the word "resolution" already has another meaning in the field of automatic theorem proving, so I shall think of steps 3 and 4 as *discovery*. Strong detailed models of proof structures must be known which can guide the whole process, otherwise we risk unbridled speculation and wild goose chases. Such proof structures are determined by the problem areas to which they relate. It is not yet feasible for computer systems to invent such elaborate structures for themselves, they must be provided by humans. Invention may not be too far away, recent work in learning [Desimone 89] has developed techniques for learning non-sequential proof structures. With suitable structures, all of steps 1-4 can be automated. Later chapters will show this in the context of two problem areas.

MOR has wide implications for search control. Effectively, a whole subtree (corresponding to all the possible values of the meta-variable) of a search tree can be explored at once. The identity of each meta-variable may be discovered progressively, and commitment made to a particular branch only as contingent decisions create constraints. Search control is concentrated on goal-directed, structured reasoning, rather than data-directed reasoning.

Reasoning this way is *middle-out* in the sense that key decisions about values are postponed to the middle of the process, once specific information is available, directed by the relevant stage of the overall plan. This should be contrasted with conventional approaches such as *top-down*, and *bottom-up*. A top-down approach would start with a main plan and refine it into progressively smaller components, but each refinement would be of a completely instantiated object to a completely instantiated object. A bottom-up approach would pick instantiated

bottom level components, and try to assemble them into a plan. A middle-out approach will also start with a plan, but instead of refining components which describe instantiated object-level stages, it will reason with uninstantiated versions, until constraints determine the variables' identities. In principle, a middle-out approach could be attempted in a bottom-up style, but it would be more difficult to achieve enough structure to constrain variables sufficiently.

The technique of MOR is domain independent, although efficiency demands that its management be tailored to the task in hand, as for other forms of search control.

Greater flexibility brings the need for greater control. Search must be controlled to pay attention to instantiated items, which provide more information than variable ones. The use of meta-variables for higher-order object-level entities requires special attention, as will be described in later chapters.

In our domain of automatic theorem proving, proof sketches and propositions containing meta-variables can be expressed by stretching representation beyond what is allowed by the grammars of the languages of formal logic, which do not admit meta-variables. Despite this apparent departure from rigour, correctness can be ensured either by restrictions on the use of the variables, or by running the instantiated proof through an object level proof system checking afterwards.

Most automated mathematical reasoning techniques are conventional top-down or bottom-up systems. Dominated by the form of the current expression, they concentrate on choosing amongst the visible subsequent steps. Recent work has tried to make more use of strategic knowledge - understanding of proof structures and planning. MOR builds on those ideas and moves closer to incorporating some of the more psychologically-based advice of such experts as Polya [Polya 45]. I shall return to this below.

1.2 Patterns and Speculation

Structured speculation is a core mathematical activity. Mathematicians rely on their ability to perceive patterns and then select related ones from their experience and adapt them as required. Adaptations are typically formed by abstracting a general pattern and using it to provide a structure within which speculation can take place and to develop a new version of the pattern, tailored for the current problem. Existing knowledge feeds the speculation. Known theorems and their proofs suggest connections, patterns and components for new proofs.

Experienced mathematicians will analyse proofs in order to abstract patterns from them and use these to suggest more general results and generate new concepts. Lenat's controversial thesis [Lenat 82] was an attempt to capture and use their strategies for discovery of new object-level concepts.

Proof structure is vital to guiding the kind of speculation which is likely to be fruitful. Individual proof components may have their own local proof structures. A simple example is that a constructive proof of a conjunction always splits into two subproofs, one for each conjunct.

Knowledge about purpose can supply more structure. For example, decidability proofs are likely to draw on arguments about termination. Indeed structure may be imposed on a proof precisely in order to make it fit some intention. A proof of the order of complexity of an algorithm will involve introducing objects with which comparison can be made. When we generalise a theorem, the knowledge of whether or not we intend to produce, for example, the special conditions which ensure tail-recursion will influence the structure we use.

All these means can be used to suggest and constrain speculation. One can then embark on a "sketch" proof, knowing how to start and approximately how to proceed, but expecting to sort out the details on the way, as the requirements become clearer. The clarification, informally, may be something like "I can't

produce a Z unless I use X" or (after some inference) "I could try A now, let's see if that leads to B", or perhaps "Suppose I knew K ...". At the level of knowledge about the proof process, one may even conjecture that if P were provable, steps S would be sufficient.

Much of Polya's advice to mathematicians [Polya 45] tries to spark insights into structure and inspire speculation. Although commonly too high-level to be used in current automated mathematical reasoning systems, it has been invaluable to many human mathematicians. Typical suggestions are

> Can you restate the problem?
>
> Can you think of a similar problem?

In [Polya 45] he writes:

> "Heuristic reasoning is reasoning not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution of the present problem. ... We need heuristic reasoning when we construct a strict proof as we need scaffolding when we erect a building."

Pertinent to this experiment with MOR is his dictum "Examine your guess". Of this he says:

> "Guesses of a certain kind deserve to be examined and taken seriously: those which occur to us after we have attentively considered and really understood a problem in which we are genuinely interested. Such guesses usually contain at least a fragment of the truth although, of course, they very seldom show the whole truth. Yet there is a chance to extract the whole truth if we examine such a guess appropriately."

In his book "The Computer Modelling of Mathematical Reasoning" [Bundy 83], Bundy points out the potential contributions of psychology to automating reasoning, as yet comparatively unexploited in comparison to those of

logic. MOR is a limited, but interesting attempt to emulate some of mathematicians' proof sketches, assuming their structure is known, but details are still to be ascertained. I will not attempt a detailed argument on the psychological validity of this technique. It is still too early to make such significant claims, but to show that this could contribute to such developments, my concluding chapter will draw on literature from the psychology of mathematics.

Although automating speculation is desirable, it is difficult. The languages used to describe mathematics are not the same as the language used to describe the activity of doing mathematics. The various object-level languages of formal mathematics, including logic, are strict and precise. Their power has been one reason for the comparative success of automating some mathematical processes, so giving them up is risky. But escaping a little, while still allowing the mathematical language to make suggestions, is attractive. That is the choice of MOR.

This kind of problem is common in Artificial Intelligence research. The question is, "What sort of meta-language should be used to describe the activity, and how should it relate to the object language?". The solution explored in this thesis spans issues both of language and of search control.

1.3 A Brief History of Automated Mathematical Reasoning

Philosophically, there are differing opinions on what mathematics is, and how it relates to mathematicians and the rest of the world. Platonists regard mathematical objects as existing, immaterially, in their own terms, independently of humans. Intuitionists view it as the product of our design, inextricable from the world. Although this affects their beliefs about the meaning of mathematics, and the operations it is valid to use within it, in many contexts it has comparatively little effect on the style of that operation. Whatever standpoint is taken, formal mathematical proofs still function in terms of their own symbols.

For classical logic, Herbrand's theorem shows that the provability of a quantifier-free formula of predicate calculus is consequent only on the language of the formula. Proof is thereby separated from external notions of truth. The possible number of proofs which might need to be considered is much reduced.

This crucial result has made it possible to concentrate on the symbols and the rules for manipulating them. Once computers became available, the initial temptation was to use their considerable powers of symbol-crunching to try out all the possibilities. Even using standard procedures such as normalisation [Gilmore 60], [Davis & Putnam 60], it rapidly became clear that this was neither adequate, efficient nor elegant.

Improvements were sought by refining the standard procedures for managing logical expressions. The resolution rule [Robinson 79] arose from this approach, and was further refined in the multitude of work on uniform proof procedures. Uniform, that is, with respect to the nature of the proposition to be proved and its associated theory. Such refinements exploit purely structural logical properties, such as whether all sentences correspond to Horn clauses. They are not tailored to theories or to proof patterns, unlike automated reasoning systems developed later.

Such uniform procedures are highly data-driven, operating in single steps determined by the current state of the expression being worked on. The search inherent in such tasks can still be huge, and so attention focussed increasingly on what strategic guidance could be given.

A solution to these search problems was attempted in systems like PRESS [Bundy & Welham 81] and the Boyer-Moore theorem prover NQTHM, [Boyer & Moore 79], [Boyer & Moore 88], which were developed after careful examination of the structure of solutions to, respectively, solving algebraic equations and proving theorems involving induction. The resulting implementations were designed around domain-specific structural guidance.

CLAM [Bundy *et al* 88],[Bundy *et al* 90a] consolidates and extends the lessons of using strategies designed from domain knowledge. It is a shell for develop-

ing custom-built and general purpose strategies. The Mathematical Reasoning Group at Edinburgh has successfully used CLAM for inductive theorem proving. The group has pioneered the use of *proof plans* to achieve desired proof structures. These plans, built around methods which perform regular components of proof, implement strategic knowledge about proof structures. Chapter 5 contains a detailed account.

In contrast, the guidance we might expect from studying the psychology of mathematicians' processes has been little used, as has already been stated, since it is so high level and allusive.

MOR tries to incorporate some of these more speculative, allusive pieces of guidance, supported by our existing powerful domain-guided strategies.

1.4 Tail-Recursive Optimisation

There are a number of well-known types of optimisation, of which tail-recursive optimisation is one. It permits compilers to avoid the building of a stack of calls when a procedure is defined in terms of itself. This is possible if the result of each call completes the execution of the function which called it. The number of recursions is unchanged, but there is a saving in space. Indirectly, of course, that results in a time saving, since less time is spent managing a smaller stack.

The development of general techniques for optimisation, as opposed to individually honing each program procedure, has been a valuable development in computing. Optimisation of code is desirable for efficiency, but potentially difficult, as the translations required may be tricky and the result unintuitive. This brings with it the possibility of a number of types of error:

- attempting an optimisation when it is inappropriate;
- attempting the wrong optimisation;
- making mistakes during the optimisation conversion;

- making mistakes when the more opaque optimised code is subsequently modified.

It is desirable that programmers write clear maintainable code, while relying on machines to take on the overhead of transparent conversions for efficiency. Machines can only achieve this through applying general procedures.

Tail-recursive optimisation is an interesting task for experimenting with *MOR*, as, in the problems I attempt, it involves reasoning about meta-variables representing higher-order object level entities, and is essentially a program synthesis problem. The programs-as-proofs principle means that program synthesis can be realised by theorem proving in constructive type theory, as will be described in chapter 3. In such systems as Oyster and NuPRL (see chapters 2 and 3), an executable program can be produced corresponding to a synthesis theorem.

The key features of this work were the use of a characterisation of proof structure corresponding to tail recursion, and the implementation of a system to find proofs with this structure automatically using *MOR*.

Using the programs-as-proofs principle, and given a standard recursive specification, a new program is synthesised using a general plan of a proof structure which will yield a tail-recursive program. Realising this plan involves speculation about a function to build the computation so far onto an accumulator. The meta-variable representing this speculation forms part of the definition of the new program, and represents something which is higher-order at the object level. This approach is unlike transformational optimisations, where the existing steps are re-grouped and re-ordered.

1.5 Generalisation

Generalisation lies at the heart of mathematical discovery. It is a powerful tool with a variety of rôles - amongst others, it may be used to:

- define new concepts (in the limited sense of extending or recombining existing ones),
- turn proofs developed around specific examples into ones valid for ranges of examples,
- produce more readily provable versions of true, but less readily provable formulae.

These activities require the vision and grasp of structure that comes with experience, otherwise they are liable to lead to pointless or impossible notions. Nonetheless, with such versatility, it is not surprising that generalisation is a common device.

Generalisation's suitability as a candidate for experiments with $\mathcal{MOR}$ is clear, particularly for the second and third of the rôles described above. These are speculative mathematical activities, still with an obligation to provide a constraining structure.

We see generalisation's outcomes in terms of more powerful results, but rarely the process which leads to the formulation of these new theorems. Van der Waerden's account of a lunchtime brainstorming proof of Baudet's conjecture [der Waerden 71] is an unusual example of this. His description brings out one surprising feature of generalised theorems - in the context of induction proofs they may be easier to prove than their "simpler" originals. The reason for this is that induction gives us an extra hypothesis very similar to the goal to be proved. If we strengthen the goal by generalising it, we strengthen the induction hypothesis too. This was part of the insight which led to my choice of problem class.

To cover the diversity of all possible generalisations would be unmanageable, not least because there are so many alternative generalisations of any one theorem. Tail-recursive optimisation is, in fact, a type of generalisation. The success of the representation technique used there suggested that it might be more widely applicable.

Initially, I studied those classes of problems which are standard generalisation problems in inductive theorem-proving. In all of them, it is clear that generalisation has a particular rôle, that of enabling the use of the induction hypothesis to complete a section of the proof. Eventually I saw how it was possible to make this enablement of induction the guide, by using MOR as a speculative device to explore the generalisation of segments of the theorem obstructing a successful inductive proof. This is in contrast to taking the usual approach of noticing repeated terms in the original theorem.

Taking induction-enabling as a motivator turned out to be a unifying and fruitful approach to controlling the instigation and identification of successful generalisations. By doing this, a number of apparently separate classes of generalisation could be dealt with within a single approach. This approach subsumed existing generalisation techniques, and could also deal with problems outwith their domain of relevance. The change of motivation provided an analysis of inductive proofs requiring generalisation. Chapter 9 describes this.

1.6 Implementation Decisions

The key components of my experiments with MOR were plans for known proof structures and the use of higher-order unification for MOR. It was necessary to decide whether to implement a proof-planning system on top of a higher-order logic-programming system, such as λProlog, or make higher-order unification available within the existing CLAM proof-planning system (described in chapter 5) and its underlying constructive logic proof development system, Oyster (described in chapter 3). The latter had some clear advantages:

- The essence of the technique is in describing strategies to achieve proof structures. This is exactly what CLAM is for. The meta-variables are a device to enhance this.

- The power of higher-order unifiability was only needed for a restricted number of specific cases, whereas most of the processing was regular programming which could take place in ordinary Prolog. Also, the proof-planning system would run extremely slowly if it had to run entirely in λProlog.

- By virtue of its basis in a constructive type theory, Oyster is capable of supplying executable programs for the tail-recursive synthesis proofs.

Consequently, CLAM/Oyster was chosen as the primary vehicle, and higher-order unification was made available through augmenting CLAM's repertoire of available predicates.

1.7 Outline of Thesis

Although the *MOR* representation used is similar in both my work on tail-recursion optimisation and that on generalisation, the nature of control information used differs. The bodies of work comparable to each of them have little in common with each other. Where appropriate, I shall deal with the two topics separately.

In chapter 2 I give a broad review of related work. Chapters 3 and 4 present relevant theoretical work on logic and type theory, and on higher order unification. The basic CLAM proof-planning system is described in chapter 5. The account of my own work starts after that, beginning with the implementation of *MOR*, in chapter 6, and followed by its use for tail-recursive synthesis and generalisation in chapters 7 and 9. Chapters 8 and 10 compare this research to other work addressing similar problems. Lastly, I draw conclusions and discuss further work which could be pursued in chapter 11.

Chapter 2

Context and Review of Related Work

The purpose of this chapter is to give some background about related work. Detailed comparison with the work described in this thesis will be addressed in chapter 8. As well as introducing some basic notions, this chapter will provide context for my work. Relevant work is in a number of areas:

- the general area of theorem proving applied to similar types of problem. Since my work is intimately bound up with the control of the proof process, it is important to describe this control for comparable systems, albeit briefly;
- algorithm and program synthesis;
- generalisation;
- unification;
- meta-variables, where the term *meta-variable* is intended to mean specifically, the use of a variable at the meta-level to represent an object level entity.

2.1 Control of Automatic Theorem Proving

Automatic theorem proving has made great progress over the last decades. The approaches are diverse and the extent of the automation varies. Some, such as Otter [McCune 89], are based on refinements of resolution techniques. NuPRL [Constable *et al* 86] and Boyer & Moore's system, NQTHM [Boyer & Moore 79, Boyer & Moore 88], are intended to form interactive systems, with macros of inferences controlled by program code (tactics) completing sub-proofs of arbitrary complexity. Such tactics may be more or less automatic, depending on their prior guidance from the user. The domain over which the proofs are to be completed is used varyingly to supply heuristics. Gelernter's system [Gelernter 63] supplied a model, which was used to prune the search space.

Although most automatic theorem proving has been based on first-order predicate calculus, some higher-order systems exist, such as Gordon's HOL [Gordon 88]. Paulson's Isabelle [Paulson 86] is also higher-order, and can handle a variety of object-level logics. More recently the Logical Frameworks [Harper *et al* 87] was designed to host work on a variety of logics. LCF [Gordon *et al* 79] and NuPRL each use special purpose logics designed for their links to computable functions. It is beyond the scope of this thesis to attempt a general account of theorem proving, and I will concentrate on those aspects and systems most relevant to the work reported here.

Without intending to define a clear division, it is worth distinguishing between automatic theorem proving and a computerised logical system with some automated components. Many theories have decidable portions. It is valuable to identify these and build decision procedures for them, for use as adjuncts to the proof process. There still remains the decision as to when to use them. Most interesting areas are undecidable, so for them we must use heuristic processes to attempt automatic proof, including the use of any available decision algorithms. In the context of interactive systems, however sophisticated the heuristic and

non-heuristic systems are, there is the possibility of expecting user control to invoke a lower level module.

Consequently, theorem proving can be seen to be working at different levels to complete proof segments - decision procedures, controlled by automated heuristics, which may in turn be controlled or assisted by a human, in an interactive system. If the reasoning about proof is kept explicit, a heuristic meta-level may make decisions about when and how to use decision procedures. In a wholly automatic system, the heuristic level must take full responsibility for control, as it is the highest level.

The CLAM system, which provides the vehicle for my work, is totally automated. It is described in detail in chapter 5, but I will summarise it here for the perspective that this gives on other work. CLAM is a proof-planning system, applying heuristic guidance at a meta-level. Plans are formed by combining components called *methods*. These are freestanding objects which perform standard operations in the domain under consideration. They may also be explicitly combined to form larger methods embodying strategic knowledge. Methods are represented as frames. Each one contains

- preconditions, which the planner can use to assess its applicability at any point,
- postconditions, which compute the subgoals resulting from its use on the current input, and
- an object level tactic to carry out the effect of the method.

Certain methods generate no subgoals, and are treated specially, since they cause termination of proof branches. The planner has a few search strategies, from which the human user chooses. Starting from an initial problem state, strategies guide the planner in constructing a combination of methods which give a solution, i.e. there are no outstanding subgoals. Methods' tactics can be combined in the same pattern as the methods were, to form an entire object level solution. By working at a meta-level CLAM is able to describe proofs structurally and

use explicit strategies and planning to attempt proofs completely automatically. Details of the object level, such as time-consuming well-formedness goals, are avoided until the major decisions have been made.

Although CLAM's main collection of methods is tailored largely for proofs of first-order formulae involving induction in the Oyster system, there is no inherent reason why it should not be extended to other types of theorem and logic. Indeed others have adapted it in this way [Wiggins 90].

As my work extends this system for particular types of theorem, as opposed to underlying logics, it is appropriate to relate it to other systems with comparable applications. i.e. the automated components of other induction theorem provers addressing similar problems.

2.1.1 Boyer & Moore's Theorem Prover

Part of the original inspiration for the CLAM system was to rationally reconstruct the Boyer & Moore theorem prover, NQTHM an essentially object level system, described in [Boyer & Moore 79] and [Boyer & Moore 88]. NQTHM is capable of proving a large and varied collection of theorems requiring induction in such domains as number theory and the LISP language itself. It operates on a first-order logical system and is capable of expressing the basic datatypes of LISP, such as numbers and lists, along with definitions of their construction, from which it defines well-founded induction schemes.

NQTHM is a cumulative system, which proves theorems which are then added to the body of rules available for rewriting, user-labelled according to the purpose for which they may be used. It is constructed around a list of what they call "methods" but in CLAM's terminology are tactics. These are attempted on the current proof goal in a "waterfall". In this the methods are tried sequentially until one applies, whereupon the whole process starts again on the resulting goal(s). No other explicit organisation to form strategies is available.

Strategic knowledge is implicit, in that symbolic evaluation is the first in the list, along with other simplifying techniques, but induction and generalisation,

complicating techniques, are among the last. A great deal of effort goes into the use of these expensive techniques to apply them wisely. A complex algorithm determines which variable induction should be performed on, and what induction scheme should be used. Generalisation is much hedged about with conditions, as I will explain later.

The lack of explicit strategy makes it difficult to detect whether any progress is being made. The system may be getting into a loop of generalisation and induction, or even generalising to a non-theorem.

NQTHM is a very powerful theorem prover capable of proving thousands of theorems without human interaction, although it has an interactive mode. It can automatically prove some complicated verification theorems. However, it does not address synthesis, as all variables are assumed universally quantified, and there is no mechanism for existential quantification, which is required for synthesis.

2.1.2 LCF and Related Work

LCF stands for Logic for Computable Functions [Gordon *et al* 79]. It is an interactive system for proof development, based on classical, not constructive logic. Consequently it does not embody the *propositions-as-types* and *proofs-as-programs* principles. Its style differs from NQTHM, being primarily intended as an interactive system, but one for which *tactics* (macros of inference rule applications applied as directed by an ML program) will be available. These can be used as shorthand to handle some subgoals automatically.

It is designed especially for theories about programming languages and recursive functions. It consists of two parts:

- a logic PPLAMBDA (polymorphic predicate lambda calculus) in which properties of programs can be stated, and specific theories described in the usual way;

- the ML programming language which enables tactics to be programmed. These can be combined by tacticals - combinators which permit the construction of sequences, repetitions and branching trees of tactic combinations.

Avra Cohn used it for her automation of proofs about tail-recursive function schemata [Cohn 79] following Huet and Lang's work (section 2.2.3).

From the same stable comes the Logical Frameworks project [Harper *et al* 87]. This is an attempt to provide a grand framework capable of describing a number of logics and permitting proof development in any or across a number. Its main implementation, LEGO [Zhaohui *et al* 90], is another interactive proof development system, in a natural deduction style. It offers proof development facilities in some related type systems, which are various Calculi of Constructions.

2.1.3 NuPRL

NuPRL was built for implementing mathematics in a system of constructive logic. Like LCF, it is an interactive system for proof development, but being based on constructive rather than classical logic, it embodies the *propositions-as-types* and *proofs-as-programs* principles. It incorporates Martin-Löf type theory, like the Oyster system which is a Prolog-based version of it. So NuPRL, like Oyster, uses the Curry-Howard isomorphism between proofs and programs, to let proofs of theorems synthesise and verify corresponding functional programs.

In NuPRL, like LCF tactics of inferences can be built. There is a library mechanism so that theorems which have already been completed may be kept available as lemmas for later use. The thrust of the work is more towards demonstrating that a variety of tasks can be specified in logic and that a provably correct program to perform them derived.

Constable et al. [Constable *et al* 86] say "... although the system provides a nontrivial level of deductive support, none of our methods can presently be

described as theorem provers". The emphasis of their work has been to demonstrate the power of the system to express complex ideas in a variety of domains.

This type of control does not compare with CLAM's meta-level planning and explicit exploration of the proof tree. It operates on nodes, and tactics either succeed or fail. All arguments to inference rules must be completely instantiated before the rules are applied.

2.2 Algorithm and Program Synthesis

In this review section I shall only attempt to survey the most directly relevant work from this major subject area, i.e. that pertaining to the generation of tail-recursive equivalents of recursively described algorithms. In addition, as the work reported in this thesis is on synthesis I will not dwell on the host of material on transformation, beyond describing that which is highly relevant to the work described here. Research which is analysed in detail in chapter 8 will only be described briefly here.

There are some well understood techniques for turning recursively defined programs into iterative ones. This may involve a number of stages, converting a convoluted recursive algorithm into a simpler one, and then that into an iterative version. Some of these techniques stem from work on optimising compilers, but not all of this work is automated, by any means.

Many authors comment on the difficulty of this process where clarity may be sacrificed for efficiency. When informed user involvement is required to assist the process, such loss of clarity may be crucial, and even prejudice the success of the final outcome. Depending on the overall aims these are critical issues, especially if the eventual program is to be easily modifiable.

The common approach taken is to demonstrate that if an algorithm or program has some distinct form, usually described by a template or schema, then some given transformation will produce a tail-recursive equivalent. There are a number of component processes to be considered:

- selection of schema;
- identification of instantiations for a particular schema - involving higher-order matching and potentially choices between non-unique matches;
- proof that the schema, as instantiated, achieves the desired efficiency gain;
- justification that the synthesised program is genuinely equivalent to the original.

2.2.1 Darlington and Burstall

John Darlington and Rod Burstall are responsible for some of the seminal work in this area. Through the 70's they published descriptions of work on different approaches to program synthesis and transformation.

Darlington's Thesis

Darlington's thesis (for which Burstall was supervisor) "A Semantic Approach to Automatic Program Improvement" [Darlington 72] is about his system of refinement by stages from a high level description involving recursions of arbitrary complexity to a lower level target language. Improvement is achieved by the compilation processes which generate subsequent levels.

Only the first level is relevant to this work, that which takes a set of recursion equations over set primitives, and yields an equivalent description in an iterative language. As he points out, other researchers had described techniques for achieving these gains in efficiency by relying on schemas, but only for certain classes of algorithm, e.g. those which are linear recursive, i.e. whose recursive definition calls on the function being defined at most once, though the name of the function may appear more than once if each appearance is in separate branches of a conditional.

To embrace a more extensive array of types of problem, his system uses both the structural form of the input and knowledge about the properties of the specific functions involved to attempt two kinds of improvement:

- re-arrangement of the computation sequence to reduce stack utilisation and function entry and exit calls;

- re-arrangement of the code to avoid repeating identical calls at separate points in the calculation.

The goal is to get sound improvements rather than a complete set.

The former of these forms of improvement is relevant here, called "structure recognising". This uses schemas expressing particular recursive forms, and matches the input expression against them until a match is found.

The matching used is F-matching. I will describe this briefly before returning to my account of the system it serves. It uses an algorithm similar to first-order unification, to compare terms considered as tree structures. That is, it traverses the term trees looking for matches at comparable points, and instantiating variables where necessary. It differs from first order unification in that variables may be used to label non-leaf nodes, and it can cope with permutations of arguments. So, taking upper-case letters to denote variables, and lower-case letters to denote constants, $F(a, b)$ can F-match with $g(b, a)$. However, it will not find other more complex matches which a general second-order matcher could, such as $F(a, b)$ matching with b, where F may be $\lambda u \lambda v.v$ or $\lambda u \lambda v.b$. Nor will F-matching find other matches which span multiple nodes of the term tree, such as $F(0)$ with $s(s(0))$.

The matching is further qualified by conditions on the actual functions involved in the match. Each of the five schema patterns given is accompanied by up to three translation conditional sets. Conditions are such things as whether the function in a particular rôle is associative or has an inverse. They perform checks that the recursion is not being made *more* complex through inadvertent

choice of match for functions. The system does not have to prove all the conditions or create the witnesses for their truth. For some, such as finding the inverse of a given function, it uses a device called an "oracle" - it asks the user! Having satisfied the conditions, the translation supplies the code template to be used.

This system described in his thesis is highly automated, but there is no need for an elaborate control structure. Backtracking over the different schemas when matching or conditions fail is sufficient.

Folding and Unfolding

Later work from Burstall and Darlington [Burstall & Darlington 77] develops some of the ideas from Darlington's thesis. Manna and Waldinger came up with some similar ideas around the same time [Manna & Waldinger 74].

Techniques are described which re-arrange the order in which operations are performed in a program so that it becomes more efficient. Separate computations are collapsed into single ones. Control over this is achieved by categorising transformation operations, such as expanding a function application using the function's definition, (*unfolding*), the inverse of this (*folding*), and *abstraction* - introducing a "where" clause to derive a new piece of definition from an existing one. These categories are then used to build a combination of transformations which has the effect of improving on the original program. Different kinds of improvement are possible within a unified system. The system is automated to a considerable extent, but receives some clues from the way its task is initially set up.

Naïve to tail-recursive transformation is described, and a method to incorporate it into the automated system discussed.

Darlington pursued these ideas further still in [Darlington 81]. The goal of this research is the development of program transformation methodologies, which influences the nature of its automation and user-interaction. Although

automated components are sought, the context is one of experimentation, so that the automation can be developed.

Topics explored in this work are:

- Optimising computation of the fibonacci function by building in an auxiliary tupling function to store intermediate results and avoid recalculation;
- Naïve to tail-recursive transformation;
- Top down to bottom up recursion - an alternative way of evading the recursive problem - instead of starting at the top and needing the next value, start at the bottom, and feed the current value into the next calculation;
- synthesis of programs from definitions written in logic;
- definition of new subfunctions;

The notion of *forced folding*, a user-controlled device, is used to drive the overall combination of transformations towards the goal of folding the program back into a recursive program with improved properties.

2.2.2 Feather

Martin Feather, another of Burstall's students, followed up Darlington's work, and developed another system, ZAP, for user-controlled recursion conversion [Feather 79]. Although not using the schema approach, it is worthy of note here because of its utilisation of second-order variables and matching. He wanted a system which could help a user cope with larger, more complex examples, and which could operate at a higher level. This was especially important in order to utilise user interaction to best effect.

In pursuance of this higher level operation, he identified "tactics" (not to be confused with CLAM tactics which will be described in Chapter 5) which performed different kinds of improvement. These tactics were:

- **Combining.** This took cases of embedded functions which traversed some intermediate data structure twice (as output and then input) and converted it into a function with a single pass, but with a more complex function applied each time.

- **Tupling.** This took separate function calls which used the same argument values, and therefore risked recomputing the repeated values. It generated a new function which only called on the arguments once, and produced an output tuple consisting of each of the individual functions' outputs.

ZAP took a context including details of function names to be improved, and proceeded to transform them both automatically and with interactive control using unfolds, folds and lemma-justified rewrites. The context consisted of definitions of functions, lemmas about them, names of functions to be unfolded, names of functions to be permitted to remain in the improved definition and outlines for functions to be improved. These outlines might specify a particular case structure or recursive structure to use, the appropriate arguments for them and perhaps even the approximate nature of the answer. Along with a knowledge of the constructors in the theory, the context enabled ZAP to decide whether or not the user's requirements for improvement had been met. The context and record of interaction provided documentation of the transformation process.

When ZAP could not automatically achieve a new definition with the desired improvements, the user could supply the details. If the user's definition was wholly specified, ZAP had only to check its equivalence to the original. However, just as in the statement of context, the user had the option of indicating approximately the form the solution was to have, and leaving ZAP to sort out the details. Approximation could take two forms:

- The use of a special function variable symbol "$$" which would be permitted to match tuple and "where" constructions, constructors and constants. It could also match certain declared functions. The symbol could be used more than once in different positions in a term and have the different occurrences bound to different values.

As an example of its use Feather gives the following goal, which expresses the notion of the function *sumsquares* (defined below) being rewritable into a recursive definition:

$$sumsquares(N :: L) \Leftarrow \$\$(N, sumsquares(L))$$

where:

$$sumsquares(L) == sum(squares(L))$$

$$sum(nil) == 0$$

$$sum(N :: L) == N + sum(L)$$

$$squares(nil) == nil$$

$$squares(N :: L) == N * N :: squares(L)$$

taking *nil* to represent the empty list, "::" list construction, and == to denote definition. The goal unfolds to:

$$N * N + sum(squares(L)) \Leftarrow \$\$(N, sum(squares(L)))$$

and matching binds \$\$ to $\lambda x \lambda y.x * x + y$.

This corresponds to the "Combining" tactic above.

- The use of other function variable symbols to abbreviate some intermediate function composition, provided this did not involve any functions not permitted to appear in the eventual definition. These variables were given a unique name preceded by two ampersands to allow the definition of the new function.

  This allowed more efficient alternative recursive structures than the standard ones to be employed. An instance would be recursion down a list of characters a word at a time instead of a character at a time. The naïvely defined algorithm would find the first word, and then remove that word by

traversing the beginning of the list again. A more efficient version could be built by collapsing these functions to one whose output was a pair consisting of the first word and the list of characters following it.

As in the other type of optimisation, ZAP was able to identify such intermediate functions by maximal unfolding, normalisation and matching.

This corresponds to the "Tupling" tactic above.

Like Darlington, Feather needed second-order matching for these tasks, and describes his implementation as a version of Huet and Lang's matcher [Huet & Lang 78] with restrictions and extensions.

The restrictions were to inhibit function variables from certain matches involving function symbols which ZAP was trying to eliminate or only use in ways guaranteed to achieve desired recursive structures. The main extension was to build in associativity and commutativity.

With the move to larger scale problems, choices arose over which tactic to apply, and which expression to work on first, since the ordering of the translation process affect what improvement was reached. Feather's strategy was to work from the innermost subterms upwards. He claimed no advantage for this other than that it worked and was systematic.

His system was able to operate on large problems, as he had intended: a simple compiler and a text formatter.

2.2.3 Huet and Lang

Gerard Huet and Bernard Lang's paper "Proving and Applying Program Transformation Expressed with Second-Order Patterns", [Huet & Lang 78] builds on Darlington and Burstall's work. Taking the latter's identification of schematic rewriting systems (with constraints to ensure validity) which achieve optimisations, Huet and Lang recast this as a task in second-order unification and explored the consequences. Their work described theoretical results, but without any implementation. Indeed they indicate that this is not a major goal, in

the paper being described at least, perhaps because of the interactive rôle they anticipate in conjunction with their technique.

In their introduction, the authors note the

> "huge gap between practical software certification techniques and the theoretical tools defined for formal proofs of programs"

They suggest that

> "One way to close this gap is to write interactive systems that will help the programmer to design, debug, run and ultimately validate his system.
>
> A desirable feature of such a system is the ability to manipulate programs into various forms, while preserving their meaning.
>
> ...
>
> As a systematic approach to high-level optimization for programs, it is the natural complement to structured programming development techniques ..."

Huet and Lang extend Darlington's repertoire of schemas, by supplementing it, generalising and also by omitting schemas which are effectively subsumed by others when using their technique. They characterise the schematic rewritings in terms of *transformation templates*, which are triples consisting of

- a schematic description Σ, of a functional program f, e.g.

$$f(x) \Leftarrow \textbf{if } a(x) \textbf{ then } b \textbf{ else } h(x, f(e(x)))$$

  where a, b, e and h are all variables, typed appropriately.

- a schematic description Σ' of the corresponding optimised functional program f': for this example

$$f'(x) \Leftarrow g'(x, b)$$

where

$$g'(x,y) \Leftarrow \textbf{ if } a(x) \textbf{ then } y \textbf{ else } g'(e(x),h'(y,x))$$

These descriptions are also available unabbreviated in an ALGOL-like notation.

- any constraints $< \kappa_1, ..., \kappa_n >$ which must apply to the entities used in describing the schemas continuing this example:

$$\forall x\ h(x,b) = h'(b,x)$$

$$\forall x \forall y \forall z\ h(x,h'(y,z)) = h'(h(x,y),z)$$

$$\forall x\ h(x,\flat) = \flat$$

where $\flat$ is the base element of the type of object required for h's second argument position.

The example used is just one of several templates they supply.

They are mainly concerned with the validation of such templates, and the organisation of libraries of templates so that subsumed templates can be omitted. To a lesser extent they are interested in recognising whether a particular template is applicable to a given program, and choices involved in this process are neglected, probably because the intention is that the whole process will take place within an interactive system.

Applicability of a template is tested by using second-order *matching* of the template's first element against the program fragment. Second-order is sufficient because no variable in any of their schemas is of higher order than a function. So for a program fragment $\mathcal{P}$, they require a substitution σ such that $\mathcal{P} = \sigma\Sigma$ and that each constraint is valid under the substitution, i.e. each $\sigma\kappa_i$ is valid. Then $\mathcal{P}$ can be replaced by $\sigma\Sigma'$.

Their method is as follows:

1. Apply the matching algorithm to Σ and $\mathcal{P}$, to get a finite set of possible substitution sets. These assign the variables in Σ but not those in Σ' or $< \kappa_1, ..., \kappa_n >$.

2. Although they do not say how the choice is made between candidate substitutions, a substitution σ is "completed" to cover these other variables. This seems to mean they are given a token entry in the substitution, the details of which are supplied by the next stage:

3. The constraints (with known substitution applied) are proved. This process instantiates the remaining unknown variables.

4. P is replaced by $\sigma\Sigma'$

The various choices involved here are complicated if automation is required. The initial matching may supply more than one substitution. Not only must the system decide between these substitutions, but it must identify the variables which occur in the 'output' and not in the 'input', by using proofs of the constraints. Some constraints will easily identify variables, others won't, so the choice of constraint to prove first is significant. In the presence of unidentified meta-variables representing higher-order entities, a poor choice could be disastrous. I will give a more detailed account of this and a comparison with my own work in Chapter 8.

2.2.4 CIP Project

Helmut Partsch's book "Specification and Transformation of Programs" [Partsch 90] is a general text on methodologies of program development. From his background working with Bauer and Wössner and with the Munich CIP project, he draws on this and related sources to describe a variety of transformational techniques, some of which are for tail-recursive optimisation.

At the beginning of the book, he outlines the CIP view of the rôle of such techniques as within a software development framework leading to: "... a separation of concerns in program development between man and machine, man being responsible for creative aspects, and machine for mechanical tasks".

He lists typical mechanical tasks as:

- support for validating the specification,
- performing program transformations,
- support for verifying applicability conditions of transformation rules,
- keeping track of the development process.

and typical creative aspects as:

- formalisation of the problem,
- selection of suitable development strategies,
- selection of transformation rules and indication of respective point of application,
- derivation of new rules (if necessary), including the respective proofs of their correctness.

The CIP philosophy is comparable to Huet and Lang's. Templates are given for some transformations which will turn certain recursive program fragments into tail-recursive equivalents. Some examples of effective transformations are given without templates for more complex operations. The expectation is that a human will select the relevant transformation, and a computer will apply and check it, perhaps with some human guidance. Automation is not dealt with.

Partsch's book follows a more thorough technical treatment by Bauer & Wössner. In [Bauer & Wössner 82], they describe a number of transformations similar to Huet & Lang's, which, if applied, achieve tail-recursive optimisation. These transformations involve the usual conditions of certain of the functions involved being associative, and there is no discussion of automation. Algorithms are also given for transforming **any** linearly recursive function into a tail-recursive, **but highly inefficient** equivalent. These are credited to Paterson and Hewitt [Paterson & Hewitt 70], and involve performing the entire computation at each recursion. They compute all the values that would be stored in

an accumulator at successive steps, starting from scratch each time. Bauer and Wössner point out a variant on this which is to store the successive values using a stack, but that would seem to defeat the purpose of tail-recursion in the first place.

2.2.5 Harrison and Khoshnevisan

Harrison and Khoshnevisan's work [Harris & Khoshnevisan 88] also follows on from Darlington's, and to some extent from the CIP project. They dissect function descriptions in terms of their functional components and by using arguments about functional structures which ensure certain properties, are able to specify new equivalent functions with the desired properties, in some cases. The approach is closer to templates than the fold-unfold style.

They have developed existing analyses of linear functions to provide the automatic transformation of such functions into tail-recursive equivalents building bottom-up. Further, they can also transform such functions into tail-recursive equivalents growing top-down, as attempted in this thesis, conditional on explicit constraint equations (such as associativity) on some of the component functions. This second transformation is very similar to some of Huet & Lang's templates, and suffers from the same problem of inventing suitable accumulator initialisation values and accumulating functions which satisfy the constraints. The constraint equations given define the possibilities systematically. This is a good, thorough approach, but essentially static, without the opportunity to use other available information to create new, but currently unknown functions. The major advantage of this approach is that it makes explicit just how the function components operate to fulfil their rôles.

2.2.6 Miller

Miller's work on $\lambda-$Prolog, a higher-order extension of Prolog, has been applied to program transformation. In [Miller & Nadathur 87], he and Nadathur describe a structure which can recognise tail-recursive programs, using template

structures, and from them produce iterative programs in $\lambda-$Prolog. They also note the drawbacks of using templates to cover many different patterns of program. They point out that the following program to calculate the sum of two numbers:

$$sum(x,y) == \underline{if}\ x = 0\ \underline{then}\ y \\ \underline{else}\ sum(p(x), s(y))$$

where p is the predecessor function, is tail-recursive, following this schema:

$$Fun(x,y) == \underline{if}\ C(x,y)\ \underline{then}\ H(x,y) \\ \underline{else}\ Fun(F_1(x,y), F_2(y))$$

However, the following function to calculate the greatest common divisor of two numbers is tail-recursive too:

$$gcd(x,y) == \underline{if}\ x = s(0)\ \underline{then}\ s(0) \\ \underline{else}\ \underline{if}\ x = y\ \underline{then}\ x \\ \underline{else}\ \underline{if}\ x < y\ \underline{then}\ gcd(y,x) \\ \underline{else}\ gcd(x-y, y)$$

but it doesn't fit the schema. They state that "there is no (second-order) term whose instances represent only tail-recursive programs and also [the *gcd*] term and the term representing the *sum* program". This is only part of a wider problem, as the conditional structure could be even more complex.

They finesse this problem by returning to the definition of tail-recursion. A program F is tail-recursive if it is of the form:

- $H(x,y)$ - there is no recursive call, or
- $F(H(x,y), G(x,y))$ - purely a recursive call with modified arguments, or
- $\underline{if}\ C(x,y)\ \underline{then}\ H_1(F,x,y)\ \underline{else}\ H_2(F,x,y)$ where H_1 and H_2 represent tail-recursive programs.

Using this formulation, they can combine the three possibilities using $\lambda-$terms. Not only does this provide a recognition mechanism, but also a transformation to an iterative form. This is achieved by guessing cases. The first case above corresponds to terminating the recursion; the second, to resetting the variables of the iteration; and the third, not surprisingly, to a conditional.

Higher-order unification is involved in matching the input to this transformation.

In [Hannan & Miller 88], Hannan and Miller describe a further implementation of some other higher-order program transformers for general operations such as unfolding and permuting arguments. It is not applied to specific tasks such as program optimisation.

2.3 Generalisation

I shall use the term generalisation in its mathematical sense, not in any broader sense used in Artificial Intelligence.

Generalisation is a step which allows us to postulate a new theorem as a substitute for the one we are currently trying to prove, and then use it to justify the original. This can be done to achieve a more powerful theorem, or because it easier than the current proof. This may seem perverse, but for theorems proved inductively, it makes sense. The induction conclusion will become stronger, but so will the induction hypothesis.

One of the dangers of generalisation is that of generalising to a new statement which is not a theorem. This is all too easily done when following syntactic generalisation rules, and hard to guard against.

I will analyse generalisation in chapters 9. In chapter 10, I will look at some of the methods presented here in more detail, and compare them to my own work.

In this section I will survey some methods of generalisation which are particularly relevant to induction proofs, i.e. those relating to universally quantified variables.

2.3.1 Generalising Terms to Variables

Boyer & Moore generalise terms which occur multiple times to variables [Boyer & Moore 79]. Their motivation is to make inductions work, and they regard this step as tidying up after an induction to clear the way for subsequent inductions. Their view is that an induction may leave extraneous term structure around, and generalisation fixes this.

Given a goal of the form $\forall x.P[f(x)]$, Boyer & Moore prove $\forall y.P[y]$ where y is some new variable (square brackets here, as elsewhere, indicate that P may be a compound expression). There are various conditions on this:

- $f(x)$ must occur at least twice,
- some occurrences of $f(x)$ must be on either side of an equality or in separate literals,
- $f(x)$ must not be a variable - since they don't distinguish between different occurrences of a variable, to generalise a variable would be pointless, it would just rename all x's to y's, for example;
- $f(x)$ must not be an explicit value template - terms composed entirely of constructor symbols and variables. They say that experience shows that explicit value templates like $s(s(0))$ contain too much information, which would be lost;
- $f(x)$ must not have $=$ as functor;
- $f(x)$ must not have a destructor as its functor. To achieve a little normalisation and reduce the number of rewrite rules, a "destructor elimination"

routine converts formulae involving known destructor symbols into equivalents without them. This gets rid of division by multiplying throughout by the divisor, for example. It has high priority in their system, so they expect that it will already have decided whether a destructor should be changed.

- $f(x)$ must not be something which contains as a subterm anything in the list of candidates being amassed. The reasoning given for this is the wish to capture information from generalisation lemmas for each component which might be lost if a containing subterm was generalised first.

As an example, they would generalise[1]

$$\forall x \forall y \forall z.\ rev(x) <> (y <> z) = (rev(x) <> y) <> z \tag{2.1}$$

where $<>$ denotes the *append* function, to

$$\forall w \forall y \forall z.\ w <> (y <> z) = (w <> y) <> z \tag{2.2}$$

This is now an easy theorem to prove.

The generalisation process is as follows:

1. For each term, t, in this list of candidates to be generalised, look through all known generalisation lemmas (only three are listed in [Boyer & Moore 79]). This is a rather *ad hoc* feature, which gives them an extra hypothesis, useful for the subsequent proof.

2. The set of datatypes of the term, t, is obtained, and if it consists of only one datatype, that information giving the datatype of the object is added as a hypothesis too. This protects against the generalisation of $len(l)$ in

$$\forall l.\ len(len(l)) = len(l)$$

---

[1] Although NQTHM does not use explicit quantification, variables are implicitly universally quantified at the outside of the formula. I have chosen to make this explicit for clarity and consistency.

where it would add a new hypothesis that the variable replacing the repeated term $len(l)$ must be a number, i.e. a list of *nil*'s, in their system. This theorem is true in NQTHM just because numbers are represented as lists of *nil*'s. On generalising it to

$$\forall v.\ len(v) = v$$

it is still true if v is a list of *nil*'s, a number. Since this is the type of $len(l)$, this generalisation heuristic enforces that condition.

3. Each subterm being generalised is replaced in the formula by an unused variable name.

2.3.2 Generalising Variables Apart

Aubin [Aubin 76] concentrated on two types of generalisation, of which the first extended Boyer & Moore's work to permit the generalisation apart of variables.

Aubin used primary recursion path analysis. ***Primary recursion paths*** were paths through the terms to the leaves only through the arguments which were in the recursion positions. By recursion position, I mean the argument position on which a function is recursively defined.

To prove

$$\forall x.\ x + (x + x) \ = \ (x + x) + x \tag{2.3}$$

he proves the more general

$$\forall x \forall y.\ x + (y + y) \ = \ (x + y) + y \tag{2.4}$$

The task, here, is to take some theorem with the same variable appearing multiple times, and find a generalisation by careful differentiation of the variables, if such is available. Any multiply occurring variable with occurrences on both sides of an $=$ or $\rightarrow$ would be considered as distinguishable. Here, that would suggest the first and fourth occurrences of x in (2.3). These were good

candidates to be the induction variable, and so worth differentiating from the other occurrences.

If that failed, as it would in

$$\forall x.\ x * (x + x) = (x * x) + (x * x) \tag{2.5}$$

where it would again suggest the first and fourth x's, the analysis could consider adding other occurrences to the set to be distinguished. Possible candidates would be those occurring in recursive argument positions, but not primary recursive positions, such as the second and sixth x in (2.5). This could be awkward since the number of combinations grows fairly quickly, and to help out there was a routine to try out some values, so that false generalisations could be discarded, by use of counterexamples. To some extent, any base case is just such a simple check.

Jacqueline Castaing also attempts this kind of generalisation [Castaing 85]. She uses mismatches of induction conclusion against induction hypothesis to guide induction choices and distinguish variables apart.

Given an induction which had failed to result in the induction conclusion being an instance of the induction hypothesis after all available symbolic evaluation had been performed, she would attempt a generalisation. She would identify all the mismatches in term structure between the induction hypothesis and induction conclusion, and create distinct new universally quantified variables for all the mismatch positions in the hypothesis.

This plethora of new variables would then be reduced by identifying any which were replacing identical terms and were primary recursion variables in Aubin's terms. It would further be reduced by separately identifying all the rest. This is a minor variation on Aubin's approach.

2.3.3 Generalising Terms with Initialised Accumulators

Aubin also describes [Aubin 76] a technique for generalising terms with a constant in an accumulator position. An accumulator is an argument used to accumulate a function application's value progressively internally as its computation proceeds. An example is the second argument of *rev2* here. Given the need to prove:

$$rev2(x, nil) = rev(x)$$

where:

$$\begin{aligned} rev2(nil, l) &== l \\ rev2(h :: t, l) &== rev2(t, h :: l) \\ rev(nil) &== nil \\ rev(h :: t) &== rev(t) <> (h :: nil) \end{aligned}$$

an induction proof results in an attempt to prove

$$rev2(t, nil) = rev(t) \supset rev2(h :: t, nil) = rev(h :: t)$$

which unfolds to

$$rev2(t, nil) = rev(t) \supset rev2(t, h :: nil) = rev(t) <> (h :: nil) \qquad (2.6)$$

and there is no match on the left-hand side of the conclusion. On the right, we can use the hypothesis to substitute for $rev(t)$, giving:

$$rev2(t, nil) = rev(t) \supset rev2(t, h :: nil) = rev2(t, nil) <> (h :: nil)$$

but then the proof is stuck, and we would have to try a further induction, which would continue to be blocked on both sides in just the way this one is on the

left[2]. The preferable approach is to generalise the constant *nil* to a universally quantified variable a, say, and turn (2.6) into:

$$\forall a.rev2(t,a) = rev(t) <> a \supset \forall a.rev2(t, h :: a) = (rev(t) <> (h :: nil)) <> a$$

Aubin's approach would decide to generalise *nil* because it was a non-variable accumulator which was an argument of an induction application. *Induction applications* are those function applications which contain in the argument position on which they recurse either the induction variable, something whose outermost function is a constructor, or another induction application. Essentially this labels those arguments which are siblings in the expression tree to places which are part of the primary recursion path, and will therefore be involved with the symbolic evaluation about to happen. CLAM also makes use of this information, and it will be interesting to remember this technique when considering the CLAM account in Chapter 5.

In this example, just generalising *nil* to a results in

$$\forall a.rev2(x,a) = rev(x) \tag{2.7}$$

which is a non-theorem without further generalisation to install a appropriately on the right-hand side of the equality too. Aubin handles this by continuing to

---

[2]There is a nice parallel here with Boyer and Moore's "Productive Use of Failure" [Bundy 83, Boyer & Moore 79] to identify the need to perform induction. On attempting to symbolically evaluate a term, if the result would be a new term containing a strict superterm of the original, symbolic evaluation is abandoned in favour of induction. Here, the symbolic evaluation resulting from induction on t in

$rev2(t, h :: nil) = rev2(t, nil) <> (h :: nil)$

would produce:

$rev2(t', h' :: h :: nil) = rev2(t', h' :: nil) <> (h :: nil)$

The second of these is not a conventional superterm of the first, because the rule being used is transverse, not longitudinal. It is the transverse analog of a superterm. If there were a strict superterm test for transverse analogs, it would suggest that this is doomed to failure and an alternative should be sought.

expect generalisation to be "balanced" on both sides of an equality. He argues that the *nil* which is being generalised on one side of the equality should be partnered by another on the other side.

Additionally, in this case his counterexample checker can establish that (2.7) is not a theorem. If he can turn $rev(x)$ into $rev(x) <> nil$, he has a *nil* to generalise on both sides of the equality, but he needs a way of selecting this.

He looks at the mismatches between corresponding induction hypothesis and conclusion expressions. The left-hand sides of (2.6) are already being accounted for by the generalisation. What about the right-hand sides: $rev(t)$ and $rev(t) <> (h :: nil)$? The induction variable occurring within the mismatch suggests a problem. Can the mismatch at least be made more "local" and avoid the induction variable? Yes, by a process he calls *expansion*, putting the hypothesis' $rev(t)$ inside the same term structure as it occupies in the conclusion, in the same position: $rev(t) <> (\_)$ and choosing a value for the hole which preserves the value of the term. This is done by trial and error over the members of the appropriate type. He also has a number of specialist heuristics for dealing with special cases involving constructor symbols.

Having localised his mismatch on the right-hand side, he can now check the real relationship between the sources of mismatch, before leaping to the conclusion that they are identical. Labelling the two *nil*'s as b and c, he can take the base value of the induction variable and consider:

$$rev2(nil, b) = rev(nil) <> c$$

which easily evaluates to show that $b = c$. Although this seems trivially obvious in this case, it is not always so in more elaborate examples such as those involving nested functions using accumulators.

2.3.4 Hummel's Survey of Generalisation

For the sake of completeness I should mention that Birgit Hummel has written a comprehensive survey of generalisation techniques [Hummel 87] including

most of those described here. Using a broad definition of generalisation which includes use of any proposition including the induction hypothesis, she distinguishes different types of generalisation according to whether the generalisation is performed before or after induction, and whether it is applied to the original formula or the induction steps.

Hummel's PhD thesis [Hummel 90] on "Generation of Induction Axioms and Generalisation" is not available to me at the time of writing.

2.4 Higher-Order Unification

Unification is central to almost any computational logic, but as soon as we stray beyond the straightforward most general unifiers of first-order unification without built in theories, it becomes much less nicely behaved. Just incorporating one law about the associativity of a function over an infinite domain will cause infinite numbers of most general unifiers. Building theories into the unification algorithm was not required for this thesis, and I shall not expand upon it here. For a broad account of unification - kinds, algorithms, theoretical results, and applications, Kevin Knight has written an excellent survey [Knight 89].

My requirements were for higher-order unification over a domain which included sorts. I shall reserve the term *unification* to describe the case when variables may be instantiated in both of the given expressions, and use *matching* when variables may be instantiated in one only. Unifiability, of course, refers to the decision as to whether or not unifiers exist, without necessarily producing any or all of them.

Although I have used Huet's higher-order *unifiability* algorithm, when it is called on in a purely second-order setting, it provides the same answers as Huet's second-order *unifier* described in [Huet & Lang 78] would.

It is known that even second-order unification is undecidable in general. Algorithms are known for second-order problems which will find unifiers if there are any, but may not terminate if the two expressions are not unifiable. Even if

unifiers exist, there may be infinitely many, and the nice notion of "most general" no longer applies. Second-order matching is decidable, and as we shall see in chapter 4, it is advantageous to reduce problems to matching wherever possible.

I shall adopt the Prolog convention of using capital letters for variables and lower case ones for constants in the rest of this section.

2.4.1 F-matching

Darlington's work *e.g.* [Darlington 72] relies on *F-matching*, a restricted version of second-order unification. In this function variables may be used, but the substitution for them may only be another function symbol. So on unifying

$Z(a,b)$ and $f(a,b)$

F-matching would suggest the unifier: $\{f/Z\}$. Full second-order matching would additionally propose such unifiers as $\{\lambda u \lambda v.f(a,b)/Z\}$.

2.4.2 Second- and ω-Order Unification

Pietrzykowski gave an algorithm for second-order unification which was published in 1973. This was extended by work with Jensen to handle ω-order unification [Jensen & Pietrzykowski 76]. These algorithms are very complex, requiring five different operations to be performed at each point of comparison. Unlike Huet's unifiability algorithm [Huet 75], they are capable of enumerating all the unifiers if there are any, although they share with his algorithm the unavoidable lack of guarantee of termination if there are none. Enumerating all the unifiers has the necessary but unfortunate consequence that where there are an infinity of unifiers, the algorithm will also not terminate.

Huet's unifiability algorithm over terms of the ω-order simply-typed λ-calculus is more straightforward, has a further simplification when η-reduction is permissible, and produces either unifiers when there are finitely many, or stops with a pre-unifier when there are infinitely many. This algorithm specifies the set

but does not enumerate it if it is infinitely large. This makes it more usable than Jensen and Pietrzykowski's, and it forms the basis of most subsequent work, e.g λ-Prolog [Miller & Nadathur 88]. Pre-unification is required in circumstances corresponding to variable-variable unification in first-order cases. In higher-order systems, when comparing two terms with variable functions, which are not λ-variables, at their heads, e.g.: $\lambda u.F(u,c)$ and $\lambda v.F'(c,u)$ Huet simply noted that such cases would always be unifiable, though the enumeration of all the unifiers would branch infinitely. It is of course open to anyone using such an algorithm to choose a particular unifier that fits. His algorithm is described in detail in Chapter 4.

Wayne Snyder and Jean Gallier [Snyder & Gallier 89] returned to earlier work on first-order unification by Herbrand, and later pursued by Martelli and Montanari. This was based on transformations of systems of terms which cleanly separates logical issues from procedural ones. This transformation approach is close in spirit to methods of solving simultaneous linear equations by progressively assigning variables. Their solutions are not, in general, most general unifiers. They argue that this is an elegant way of analyzing invariant properties of unification and extending it to higher-order systems. New completeness proofs are presented. Their descriptions are close to Huet's and their conclusions, although shifted to a somewhat more abstract level, confirm the value of Huet's algorithm.

In $\lambda-$Prolog [Miller 90], Miller uses a restricted form of higher-order unification. He describes it as "an extension to first-order unification that respects bound variable names and scopes". The algorithm is designed for his logic programming language, and has the properties of being decidable, and providing most general unifiers where any exist. His is an untyped system, and it gains its power by only permitting β_0-reduction, not full $\beta-$reduction. This means that a $\lambda-$abstracted function, $\lambda x.t$, may only be applied to tokens which are not free in $\lambda x.t$, such as $(\lambda x.t)(x)$. The consequence of this is that he avoids the explosive parts of the unification search space, but cannot achieve some of the unifications

we would need for speculating about generalisation. It would be adequate for the existing work on tail-recursive synthesis, as it is an improvement on F-matching.

2.4.3 Typed Unification

In his thesis on search and logic programming for the $\lambda\Pi$ calculus [Pym 90], David Pym describes a development of Huet's unifiability algorithm to deal with unifications involving variable types as well as variable terms. Instead of requiring all types to be known in advance of using the unifying algorithm, they may contain variable entities too, instantiated along with the term. His extension is limited to the types of entities that are required for the $\lambda\Pi$ calculus, which would not cover all those used in Oyster. For example Σ-types, used for existential quantification, are not included. His system is not implemented.

2.5 Meta-Variables

By using variables at the meta-level not just as placeholders with which to reason, but as parameters which only become instantiated by the demands of the proof, we can postpone decisions until enough information becomes available to make them. This usage constitutes what we call *middle-out reasoning.* This technique has proved fruitful in many aspects of controlling theorem proving. The notion of meta-variables is integral and merits a short account.

As I said at the beginning of this chapter, for this purpose, the term *meta-variable* is not intended to include **any** variable used within a meta-level inference system. It is used here purely to denote a meta-level variable which represents an entity at the object level. The meta-variable may refer to first-order or higher-order entities in the object level theory. This means that we can take advantage of the naming relationship between meta-level and object-level, and apply more flexible versions of inference rules at the meta-level, for example, than would be possible at the object level. There are no theoretical problems with this, as long as the eventual result is valid within the rigorous object-level system.

Many AI systems use meta-level inference in one way or another, for planning, expert systems etc. A comprehensive account is provided in van Harmelen's thesis [van Harmelen 89] which contains a detailed analysis. He categorises meta-level systems along a number of dimensions of which the only one relevant to this discussion is the linguistic relationship between the meta-level and object level.

Van Harmelen classifies the CLAM system as bilingual, since the meta-level language (a planner and methods written in Prolog) is kept syntactically distinct from the underlying object language. His classification is based on the levels' languages being kept conceptually separate, even if they happen to use the same language. In this case, the two languages are different, since the object language is Martin-Löf Type Theory, and the meta-level language is Prolog.

Bilingualism is a common and desirable feature of meta-level systems, as van Harmelen asserts, since it offers

1. **Suitability:** the object level and the meta-level deal with separate domains, and so have good reason to be distinct;

2. **Distinguishability:** by keeping the levels separate they can be changed independently;

3. **Explanation:** control knowledge is kept distinct for explanatory purposes;

4. **Formal correctness:** work on the theoretical foundations of meta-programming by Hill and Lloyd [Hill & Lloyd 88] attempted monolingualism, but preferred bilingualism as the only way of achieving a satisfactory theoretical account.

We can now add to these an extra advantage, since middle-out reasoning is possible with bilingualism.

2.6 Conclusions

Great advances are being made in all the fields I have described here. There is plenty of scope for finding ways of tying them together, seeing their interrelationships, and using tools from one for another.

Chapter 3

Proofs as Programs

The work for this thesis has been done in the Oyster system [Horn 88] in a logic based on constructive type theory. As well as being a proof refinement system for constructive logic, the **Proofs – Programs correspondence** means that Oyster proofs can be used for program synthesis, verification and transformation. Since different proofs of the same theorem can correspond to different program algorithms, desired algorithmic properties can be obtained by ensuring a particular proof structure.

In this chapter I will describe:

- The Curry-Howard isomorphism and the Proofs as Programs principle.
- Relevant details of the Martin-Löf Type Theory and the Oyster implementation.
- Synthesis, verification and transformation of programs via proofs.

3.1 The Proofs as Programs Principle

The concept of computation in computer science is closely related to that of inference in constructive logic. Programs and constructive proofs require us to construct actual objects such as sets, functions or numbers as evidence. We may not show that "for any given natural number there is always a greater one" by assuming the converse and proving a contradiction, we must produce a program that can compute a greater number than a given one. The deep similarity of constructive inference and computation is accounted for by the Curry-Howard isomorphism.

The Curry-Howard isomorphism is between proofs and terms of the simply typed λ-calculus. Details are given in the next section. This immediately brings us close to programming because the λ-calculus can be viewed as a functional programming language. Computation corresponds to term reduction. Initially this isomorphism was perceived syntactically [Curry & Feys 58, Howard 80] as a correspondence between proofs in the intuitionistic logic of implication and terms of a simply typed λ-calculus. Logical operators such as $\wedge, \supset$ (conjunction and implication) correspond to type operations: $\times, \rightarrow$ (cartesian product and function space). Proofs then correspond to terms of the appropriate type. By using the concept of **Propositions as Types**, propositions are identified with the type of their proofs. Since each proof corresponds to a λ-calculus term, the proposition it proves may also be viewed as the type of the term. Furthermore, when we view these λ-calculus terms as programs, the proposition may be regarded as the type of the program, in the sense of specifying its task.

This proof/type construction process will be explained in more detail later in this chapter, in the context of describing the process of proof and the extraction of programs. Martin-Löf's Type Theory comprises a constructive type theory with an associated constructive logic interpretation and extended λ-calculus/functional programming constructs.

Consequently, we can see the statement $t \in T$ in different ways:

- t is an element of a type, T
- t is a proof of T
- t is a program for the task specified by T

Oyster is an implementation of a logic close to Martin-Löf's type theory. It is a Prolog version based on NuPRL [Constable *et al* 86]. It provides a sequent calculus proof refinement system, building the corresponding λ-calculus fragment (called the extract term) at each proof node, as well as developing the proof tree. At any proof tree node labelled by some proposition, for which the proof below has been completed, the extract terms assembled from the proof below form a lambda calculus term. This constructed term is evidence that the proposition's type is inhabited (by the term), which is equivalent to it being proved. I will use the word "proof" for a sequent calculus proof, "extract term" for the corresponding λ-calculus term, and "witness" (sometimes "proof witness") for a piece of evidence, possibly assumed if it labels a hypothesis.

For a complete proof of a goal which is a specification of a program, these program fragments will amount to a runnable program. If the problem has been suitably formulated, Oyster can execute this program. As some proofs correspond to verification rather than synthesis, the corresponding λ-calculus term may not always be executable.

Constructive logics may be defined as natural deduction systems, but a sequent calculus, as is used in Oyster and NuPRL, has the advantage of being modular - any node has all the information relevant to its proof present explicitly. There are no links to indefinitely distant assumptions which must be discharged later. The hypothesis list contains any assumptions, labelled with their associated proof objects.

3.2 Constructive Logic

In any logic, proofs can build on other proofs. In a constructive logic, though, a guarantee of the ability to construct proof witnesses is integral to the notion of what constitutes a proof, in contrast to classical logics using Tarski semantics. Working in a constructive logic, proof requires evidence. The evidence may be a construction based on other proofs, an actual witness or a recipe for the construction of any objects whose existence is asserted. As an example of an "actual witness", if we wished to prove that there were some natural numbers, $s(0)$ would be an "actual witness".

For example, to prove a conjunction, it is necessary to prove each conjunct. The evidence for the conjunction is the pair consisting of the evidences for each conjunct. The inference rules are designed so that the process of proof can construct the necessary evidence at each stage, and assemble the proof components from each node into the evidence for the whole proof.

This need for evidence means that we may not prove a proposition by assuming its negation and then deriving a contradiction, or by assuming in general that either a proposition or its negation must be true. If we wish to make such an assumption, we must provide evidence for it.

As in classical logic, there may be many proofs of a proposition. As already stated, a proposition A can be taken to represent the type of its proofs. The proof we construct will be one element of this type. A false proposition has no proofs, so it corresponds to the appropriate empty type, *void*.

The Oyster versions of inference rules are called refinement rules because as the proof is constructed backwards from the initial statement, it is refined into (usually simpler) components. Each rule has a corresponding construction function associated with it, for building its evidence from the evidences associated with its supporting proofs. Each use of an inference/refinement rule extends the fringe of the proof tree and fleshes out the extract term a little more, leaving

holes for the rest of the subproofs to fill. The holes will be filled by their evidence constructions. The leaves are one of the following:

- Axiomatic equalities such as $\vdash 0 =_{pnat} 0$. As I shall explain in more detail later, equality is defined with respect to the type of the expressions being compared. For Peano natural numbers (*pnat* in Oyster), $0 =_{pnat} 0$ is axiomatically true. The extract term for this is just *axiom*.

- Well-formedness goals such as $\vdash 0\ in\ pnat$, again the extract term for this is just *axiom*

- Sequents which are immediately true because the conclusion is already directly a hypothesis, and its proof construction is available, e.g. $A \vdash A$. The extract term for this is the proof construction witness.

3.2.1 Refinement of Conclusions

It is interesting to look at the Curry-Howard isomorphism in detail for a variety of common inference rules, to see how the proof construction takes place which doubles as a functional programming language. I will use P_A and P_B to represent proof witnesses of propositions represented as A and B. The proof witnesses correspond to the extract terms for A and B. For each inference rule, I will describe what constitutes proof, what the witness construction term is in Oyster, and what the type of such terms is. Table 3–1 summarises these rules. The final column is the Oyster refinement rule which would be used. For most of the rules the context makes their operation unambiguous, but some of them must be parameterised, as described next. I have not shown all the subgoals from each inference in table 3–1, as the well-formedness subgoals would make it cluttered.

- To prove $A \wedge B$ we must prove A and we must prove B. A proof of the proposition $A \wedge B$ consists of a pair $P_A \& P_B$ (written (P_A, P_B) in usual notation). The cartesian product $A \# B$ is then the type of all such proofs.

- To prove $A \vee B$ we must prove A or B and state which one, hence the parameterisation in table 3–1. A proof of $A \vee B$ consists of an indicator (*inl* or *inr*) of which disjunct has a proof, along with the proof (P_A or P_B). The proof evidence construction is $inl(P_A)$ or $inr(P_B)$. The set of all of these corresponds to the disjoint union[1] of A and B viewed as types, $A \backslash B$.

Sequent	Construction	Type	Subgoals	Refinement Rule
$\vdash A \wedge B$	$P_A \& P_B$	$A \# B$	$\vdash A$ and $\vdash B$	intro
$\vdash A \vee B$	$inl(P_A)$	$A \backslash B$	$\vdash A$	intro(left)
$\vdash A \vee B$	$inr(P_B)$	$A \backslash B$	$\vdash B$	intro(right)
$\vdash A \supset B$	$\lambda(x \in A).P_B(x)$	$A \rightarrow B$	$a \in A \vdash B$	intro
$\vdash \neg A$	$\lambda(x \in A).P_{void}$	$A \rightarrow void$	$a \in A \vdash void$	intro
$\vdash \exists x \in T.A(x)$	$(t, P_{A(t)})$	$t : T \# A$	$\vdash A(t)$	intro(t)
$\vdash \forall x \in T.A(x)$	$\lambda t.P_{A(t)}$	$t : T \rightarrow A$	$t \in T \vdash A(t)$	intro
$P_A \in A \vdash A$	P_A	A	none	intro

Table 3–1: Introduction Refinement Rules

- To prove $A \supset B$ we must show how a proof of B can be constructed given a proof of A. An arbitrary new symbol a is created, corresponding to the witness for A which will be assumed to exist. A proof of $A \supset B$ consists of a function which applied to a proof of A produces a proof of B; that is, a function $\lambda x.b(x)$ which may be applied to P_A. The type of all of these is the type of functions from type A to type B, $A \rightarrow B$. It is a special case of the class of dependent functions, which correspond to proofs of universally quantified formulae.

- Proving $\neg A$ means that we want to show there can be no proofs of A, i.e. that it is the empty type, *void*. $\neg A$ is identified with $A \rightarrow void$, the type of

[1]Set union without collapsing common elements

functions which, given a proof of A, returns a proof of some absurdity, such as membership of the empty type. This makes it similar to the previous case.

- A proof of $\exists x \in T.A(x)$ requires that we demonstrate the construction of an object t of type T, and a proof of $A(t)$. The proof construction is a pair $(t, P_{A(t)})$, where $P_{A(t)}$ is a proof of $A(t)$. The type of proofs is the disjoint union of the $A(t)$ types, indexed by t's: $t : T\#A$. This is a dependent product, because the type of the second element of any pair in it is dependent on the first element.

- A proof of $\forall t \in T.A(t)$ requires us to demonstrate that for any member t of T, we can produce a proof of $A(t)$, thus a function which takes an object t of type T, and constructs a proof $P_{A(t)}$ of $A(t)$, i.e. $\lambda t.P_{A(t)}$. The type of proofs is the type of functions from T to A, indexed by t's: $t : T \rightarrow A$. This is a space of dependent functions, because the range type may depend on a domain element.

Here is an example proof, of $A \wedge B \supset A \vee B$ as an illustration of some of these rules. Since everything must be typed, A and B are declared to be in $u(1)$, the type of basic types. This will be explained in more detail in section 3.4. To simplify the presentation, I have left out the well-formedness goals which would arise and be resolved trivially. The proof will be completed in the next section.

$$\vdash \forall A \in u(1)\, \forall B \in u(1)\, A \wedge B \supset A \vee B \quad \text{(i)}$$

$$A \in u(1) \quad \vdash \forall B \in u(1)\, A \wedge B \supset A \vee B \quad \text{(ii)}$$

$$A \in u(1), B \in u(1) \quad \vdash A \wedge B \supset A \vee B \quad \text{(iii)}$$

$$A \in u(1), B \in u(1), h \in A \wedge B \quad \vdash A \vee B \quad \text{(iv)}$$

i: This top sequent is refined by a $\forall$ introduction. This can be viewed as either the hypothesis of some new free variable over which to continue proof, or as part of the construction of a witness. The partial construction is $\lambda A.\_$, where the underscore will be filled in by the next refinement.

ii: This is similar to the last refinement. The assembled construction is $\lambda A.\lambda B.\_$.

iii: To prove an implication, we may assume a proof of the antecedent and use it. In constructive terms, we construct a function which applied to h, the witness, produces a proof of $A \vee B$, that is $\lambda h.\_$. The assembled construction is $\lambda A.\lambda B.\lambda h.\_$

Since the proof system is one of refinement, and operates backwards, these rules, called **introduction** rules, actually remove connectives from the formula being proved. This makes sense if the proof is considered in the forwards direction, when the connective is being **introduced**. Similar remarks apply to the **elimination** rules, described next.

Programming Constructs

The proof constructors are components of a functional programming language. So far we have function abstraction, the creation of tuples and canonical terms (using *inl*,*inr*). Other functional language components needed will be rules for term evaluation and accessing arguments of terms. They will be presented in the following sections. This chapter by no means describes all Oyster's rules, but it covers enough to give a clear picture. For a complete list see [Horn 88].

3.2.2 Refinements of Hypotheses

In the previous section, operations on formulae were described from the point of view of proving them, using the refinement rules to **introduce** their connectives in the conclusion formula. Table 3–2 summarises the related rules which apply when the connectives occur in formulae in the hypotheses. In that context, the effect of the refinement is to access the components which form part of the construction of that hypothesis' proof. The major connective is "**eliminated**". Just as $\wedge$-introduction creates a pair, $\wedge$-elimination splits an assumed proof of a

conjunction to yield the proofs of the pair of conjuncts. These are added to the hypothesis list. Continuing the example shows this:

$$A \in u(1), B \in u(1), h \in A \wedge B \quad \vdash \quad A \vee B \quad \text{(iv)}$$

$$A \in u(1), B \in u(1), h \in A \wedge B,$$
$$h_A \in A, h_B \in B, h_{A \wedge B} \in h =_{u(1)} h_A \& h_B \quad \vdash \quad A \vee B \quad \text{(v)}$$

$$A \in u(1), B \in u(1), h \in A \wedge B,$$
$$h_A \in A, h_B \in B, h_{A \wedge B} \in h =_{u(1)} h_A \& h_B \quad \vdash \quad A \quad \text{(vi)}$$

iv: This is the last sequent of the proof segment from the last section. To split the conjunction requires an elimination refinement on the conjunction hypothesis (**elim(h)**). In proof terms, new hypotheses are added, for the individual conjuncts. In construction terms, new witnesses are added corresponding to the components, and a linking witness, $h_{A \wedge B}$, noting the composition. The assembled construction is $\lambda A.\lambda B.\lambda h.spread(h, [h_A, h_B, P])$, where P is the extract term that will be supplied by the proof below this node. Here, *spread* takes a pair (first argument) and a list (second argument) specifying two labels and a term which may include them; on execution the function returns this term with the labels substituted by the elements of the pair.

v: The step from this sequent to the next one is an introduction. To prove $A \vee B$ we must prove one of them and state which one. If we pick A, the refinement rule is (**intro(left)**). $\lambda A.\lambda B.\lambda h.spread(h, [h_A, h_B, inl(\_)])$ is the assembled construction.

vi: The last step is to notice that the proof can be finished since the hypothesis list contains the conclusion proposition, with its witness, h_A.

That last step also completes the construction corresponding to the whole proof: $\lambda A.\lambda B.\lambda h.spread(h, [h_A, h_B, inl(h_A)])$. As we would expect, it is a func-

tion which splits a pair into its components and uses one of them to show that the disjoint union of the types is non-empty - equivalent to constructing the proof of the disjunction.

Here is an explanation of the various refinement rules applying to hypotheses, it is summarised in table 3–2.

Sequent	Construction	Refinement Rule
$h \in A \wedge B \vdash C$	$spread(h, [h_A, h_B, P_C])$	$elim(h)$
$h \in A \vee B \vdash C$	$decide(h, [h_A, P_{C_A}], [h_B, P_{C_B}])$	$elim(h)$
$h \in A \supset B \vdash C$	$P_C\{h(P_A)/h_B\}$	$elim(h)$
$h \in \exists x \in T.A \vdash C$	$spread(h, [h_T, h_A, P_C])$	$elim(h)$
$h \in \forall x \in T.A \vdash C$	$P_C\{h(t)/h_{A(t)}\}$	$elim(h, on(t))$
$h : void \vdash C$	$any(h)$	$elim(h)$

h_Z names a proof of a hypothesis Z.

P_Z is the proof construction resulting from a proof of Z.

$T\{p/q\}$ means the substitution of p for q in term T.

Table 3–2: Elimination Refinement Rules

- To prove C assuming we have h, a proof of $A \wedge B$, we may need to access the individual proofs of A and of B. In a constructive proof, we may assume these exist. This refinement just identifies h as a pair, and produces its elements for use.

- To prove C using the assumption $A \vee B$ we must produce two proofs, one assuming A, and one assuming B. The construction added is the *decide* term indicated in table 3–2. According to whether h is of the form $inl(\_)$ or $inr(\_)$, the *decide* function will yield its second or third argument, when run as a program.

- To prove C assuming h, a proof of $A \supset B$, we may split the proof into a proof of A, and a proof of C assuming B. Since h is a function which produces a proof of B given a proof of A, its application to P_A is an actual proof of B, and is substituted for the label used in the proof of C.

- To prove C assuming $\exists x \in T.A$ we may need access to the object h_T which exists and the proof that it satisfies A. These necessarily exist in a constructive proof. This refinement just identifies h as a pair, and produces its elements for use.

- To prove C assuming $\forall x \in T.A(x)$ we may want the proof instance for some particular element t of T. The refinement allows us to specify such an element (we are required to justify that it is an element of T), and provides us with a proof of $A(t)$, since h is a function which yields precisely such objects.

- To prove C assuming $h \in \mathit{void}$ is to assume something constructive logic accepts as a contradiction, inhabitation of an empty type. If we can assume that, then we can prove anything. The extract term, $\mathit{any}(h)$ reflects this.

These rules add some more functional constructs to the language: accessing components of tuples, a decision function (like an IF...ELSE...), and substitution of values for parameter names.

3.2.3 Case Splits

Although in constructive logics the proposition $P \vee \neg P$ is not generally true for an arbitrary proposition, P, it is sometimes true for certain P's.

For example, $(x =_{\mathit{pnat}} y) \vee \neg(x =_{\mathit{pnat}} y)$ is true, because a decision procedure for testing it can be described. The two proof branches would deal with the cases where $x =_{\mathit{pnat}} y$ and $\neg x =_{\mathit{pnat}} y$, and have corresponding program pieces $P_{x=y}$ and $P_{\neg x=y}$, respectively relying on a hypothesis h that x was or was not equal to y.

The refinement rule for this particular case split is called *decide*[2] in Oyster, and the program component is $pnat\_eq(x, y, P_{x=y}, P_{\neg x=y})$, a function which checks the equality of its first two arguments and depending on the result returns the 3rd or 4th argument. When this function is executed, the hypothesis label h will be substituted by a correctly typed assumed truth, as actual witness. If $x =_{pnat} y$ the substitution is $axiom$. Otherwise, $x \neq_{pnat} y$ or rather $x =_{pnat} y \rightarrow void$. To preserve the type constraints, h's substitution must correspond to a function which also yields $axiom$ but ranges over the type $x =_{pnat} y$ (which is empty), e.g. $\lambda\_.axiom$.

Sequent	Construction	Refinement Rule
$x \in pnat$, $y \in pnat \vdash C$	$pnat\_eq(x, y, P_{x=y}\{axiom/h\}$, $P_{\neg x=y}\{\lambda\_.axiom/h\})$	decide(x=$_{pnat}$ y)

The *decide* rule corresponds to having a hypothesis which is such a disjunction, and splitting the proof at that point, with each subproof having one of the disjuncts as a hypothesis. The disjunction need not literally be present in the hypothesis list for this to apply.

Different decision rules are built into the system for the different types, and there are corresponding extract term constructions, their equivalents for *pnat_eq*.

3.2.4 The Cut Rule

The "cut" rule, called *seq* in Oyster, is a common inference rule in sequent calculi. The proof divides into two branches:

1. a proof of A, the new proposition we wish to assume, from the current hypotheses, and

---

[2]Not to be confused with the *decide* function which may be constructed as part of an extract term

2. a proof of the original sequent, with A as an extra hypothesis.

The latter proof will be of the form $\lambda h_A.P_C$, a function which produces a proof of C given a proof of A. The associated construction for the *seq* node is the application of this function to P_A, the proof of A generated from the other proof branch.

Sequent	Construction	Refinement Rule
$\vdash C$	$(\lambda h_A.P_C)(P_A)$	$seq(A)$

This allows new hypotheses to be added to a proof, perhaps corresponding to lemmas, or a generalisation of the theorem being proved. Although cut can be shown to be unnecessary for certain theories, it is required for others, and can make proofs clearer.

In programming terms, it is like parameterising a function over another function or introducing an auxiliary program.

3.3 Substitution

A special inference rule performs substitution of one term for another. This splits the proof into two branches, one checks that the replacement term is equal to the original, the other proceeds to prove the original goal, with the substitution applied. No work relevant to the extract term is done by the substitution, so the extract term corresponding to the original goal is just the extract term for the latter of the two proofs below it.

3.4 Datatypes

Just as different theories may be axiomatised in classical logic, various base types may be defined for a constructive logic system. In Oyster there are integers, natural numbers, atoms (strings of characters), and the empty type *void*. This is essential to make the system useful for mathematical theorem proving or program development. Axioms associated with these types, such as their induction principles, are built in.

More complex types such as products, unions and function spaces, are constructed from the base types, as described previously. Lists of objects of named types such as integers, *int list*, can be created with the *list* constructor. More general recursive types, such as trees, can also be built using the *rec* constructor, which is described in more detail in [Horn 88].

For each type, we must know:

- How to form canonical elements: for the natural numbers, there are two canonical constants, 0 and s, the successor function. $s(n)$ is a canonical element if n is a natural number. We do not require that n be canonical, it is enough for it to be well-formed, since that guarantees that it can be evaluated into a canonical form.

- When two canonical elements are equal: in the case of the naturals, $0 =_{pnat} 0$ and $s(n) =_{pnat} s(m)$ if $n =_{pnat} m$.

Equality is always relative to type.

When terms are evaluated as part of a reduction or while executing an extract term, lazy evaluation is used. An element is defined to be canonical if its dominant functor is a canonical constructor as defined for its type e.g. $s(0+s(0))$ (function definition, such as $+$, will be explained shortly). Otherwise it is non-canonical, and may be reduced by use of reduction rules - for example $0 + s(0)$

is non-canonical. No reductions apply to a term dominated by a canonical constant, since lazy evaluation only evaluates when necessary.

If it becomes necessary to decide whether an equality holds, the rules for the type will define how that may be done. Some evaluation may then be necessary to produce canonical terms on each side of the equality, so that they can be assessed according to the rules for canonical objects.

Once formation and equality is defined for base types, it is defined inductively for the types constructed from them.

The proof construction of a statement $a =_{Type} b$ is just *axiom*. The system checks whether the terms are syntactically identical modulo renaming of bound variables, and that they satisfy the type constraints.

3.4.1 Types of Types

Since everything must be typed in a typed set theory, there is a type that these basic types and those types constructed from them using $\rightarrow, \#, \backslash$ inhabit. It is called $u(1)$ in Martin-Löf's system, and is the first of a cumulative hierarchy of universes, where $u(j)$ contains any objects which involve $u(i)$, where $i < j$. So functions from the naturals to $u(1)$ inhabit $u(2)$.

3.5 Induction and Recursion

Primitive recursive function templates for integers, natural numbers and lists are provided in Oyster. They permit the definition of primitive recursive functions in terms of the cases based on the type constructors, corresponding precisely to the structure of the type. For example *append* can be defined over the natural numbers, using the *list_ind* function constructor.

$$append(x, y) == list\_ind(x, y, [h, t, r, h :: r])$$

The left-hand side is syntactic sugar for the right-hand side. The arguments of *list_ind* are:

1. The label, x, of the argument over which the recursion is defined.

2. The value of the function when the recursion argument is *nil*, the empty list. This can be defined in terms of the other arguments (just y here), and any known constants and functions.

3. The definition of the value of the function when the recursion argument is $h :: t$, i.e. it has a head, h, and a tail, t, and :: is the list constructing function. This is a quadruple whose arguments are:

   (a) the label h,

   (b) the label t,

   (c) the label, r, representing the value of the recursive call to the function, $append(t, y)$,

   (d) the term which computes the value of the function. This can be defined in terms of the other arguments, known constants and functions, h, t and r.

This adds recursion to the functional language being assembled. Details of other recursion templates can be found in [Horn 88].

Induction is dual to recursion. Applying it as an inference rule splits the proof into step and (usually) base case branches. The corresponding constructor is just *list_ind* as described above, or an equivalent for some other type of induction. The proof step sets up subproofs which provide precisely the objects required for the function. For lists, the base and step case proofs supply their proof objects. In the step case the labels for the head, tail and recursive call are the hypotheses assuming the existence of the head, the tail and the proof of the induction hypothesis.

Sequent	Construction	Refinement Rule
$x \in pnat\ list \vdash C$	$list\_ind(x, P_{C_{nil}}, [h, t, r, P_{C_{h::t}}])$	$elim(x)$

3.5.1 Example

An example should clarify this. I will specify a *delete* function for lists of natural numbers, and then use the proof to synthesise the function. As before, I will omit the well-formedness goals, which are numerous and tedious. During the proof, at the induction step, the recursion of the *delete* function will be built. Here is the specification:

$$\forall e \in pnat\ \forall x \in pnat\ list\ \exists y \in pnat\ list$$
$$\forall z \in pnat\ member(z, x) \supset z =_{pnat} e \vee member(z, y)$$
$$\wedge \forall w \in pnat\ member(w, y) \supset member(w, x)$$
$$\wedge \neg member(e, y)$$

This requires that all members of x are either e or members of y, all members of y are members of x, and e is not a member of y. Such a specification of *delete* permits the result to be a permutation of the elements of the non-e members of x, or even have different numbers of occurrences, but that is not important here. The *append* function is defined as above, and *member* as follows:

$$member(e, y) == list\_ind(y, void, [h, t, r, h =_{pnat} e \backslash r])$$

which states that a number e is a member of a non-empty list y if it is equal to the head of the list or it is a member of the tail of the list ($h =_{pnat} e\backslash r$), and it is not a member of the empty list ($void$).

The first steps of the proof are routine. The universal quantifiers are introduced, so that we get new assumptions that there are such e and x of the appropriate types, and the conclusion refers to them. In programming terms, the universally quantified variables are the input variables over which the function will range, so they become lambda variables in its description.

1. $e \in pnat$
2. $x \in pnat\ list$

$$\Longrightarrow \exists y \in pnat\ list \forall z \in pnat\ member(z,x) \supset (z =_{pnat} e \vee member(z,y))$$

$$\wedge \forall w \in pnat\ member(w,y) \supset member(w,x)$$

$$\wedge \neg member(e,y)$$

The corresponding extract term is $\lambda e \lambda x._-$, where the _ will be filled with the extract term from the rest of the proof.

Induction on x (refinement $elim(x)$) now introduces precisely the cases we need to consider, and the components to describe them with. The proof is split into two subproofs, the base case and step case, each of which will yield an extract term. The extract term grows to $\lambda e \lambda x.list\_ind(x, \_, [x_h, x_t, x_r, \_])$, where the gaps will be filled in by the extract terms from the base and step cases respectively. This *list_ind* builds recursion into the function being constructed.

First, the base case, where x is *nil*:

1. $e \in pnat$
2. $x \in pnat\ list$

$$\Longrightarrow \exists y \in pnat\ list(\forall z \in pnat\ member(z,nil) \supset z =_{pnat} e \vee member(z,y))$$

$$\wedge \forall w \in pnat\ member(w,y) \supset member(w,nil)$$

$$\wedge \neg member(e,y)$$

The construction corresponding to an existential proof is the pair consisting of the *object* for which the theorem is true, and the *proof* that it is indeed true for that object. The theorem is true if the value of y is *nil*, so we introduce that, and then have the obligation of completing a subproof to show that the value introduced meets the rest of the specification. This is a verification[3] component

---

[3]It is worth noting that synthesis need not be accompanied by such a separate verification if no specification was made initially. A goal may just require the demonstration

of the synthesis. It is not used in executing the extract term, but ensures that the extract term satisfies the specification. Assuming that we complete this subproof, and its extract term is P_{nil}, the entire extract term grows to

$$\lambda e \lambda x.list\_ind(x, (nil \& P_{nil}), [x_h, x_t, x_r, \_])$$

Now, the step case. Here, x is $x_h :: x_t$, where :: is the function which constructs lists by joining a head onto a tail. The recursive call, x_r is effectively $delete(e, x_t)$, it labels the induction hypothesis, or equivalently, the recursive call of the function being constructed. Only if this hypothesis or some hypothesis derived from it is used in the subsequent proof to justify the conclusion, and included in the function being assembled, will that function be recursive.

1. $e \in pnat$
2. $x \in pnat\ list$
3. $x_h \in pnat$
4. $x_t \in pnat\ list$
5. $x_r \in \exists y \in pnat\ list \forall z \in pnat\ member(z, x_t) \supset (z =_{pnat} e \vee member(z, y))$

$$\wedge \forall w \in pnat\ member(w, y) \supset member(w, x_t)$$

$$\wedge \neg member(e, y)$$

$$\Longrightarrow \exists y' \in pnat\ list \forall z \in pnat\ member(z, x_h :: x_t) \supset (z =_{pnat} e \vee member(z, y'))$$

$$\wedge \forall w \in pnat\ member(w, y') \supset member(w, x_h :: x_t)$$

$$\wedge \neg member(e, y')$$

Essentially the task in the step case is to describe the choice for y according to whether x_h is e or not. This becomes the specification of most of the algorithm.

---

that a type is inhabited, without specifying any particular properties. In such cases, there may be well-formedness subproofs, but verification is vacuous.

The choice of y' for x will necessarily involve the value y for x_t, got from the induction hypothesis, x_r. Since this is an existential formula, it is a pair consisting of an object and a proof relating to that object, here the value of the function for x_t, y and a proof that that meets the specification, x_{r_y}. Using an elimination rule on this hypothesis produces these components as more hypotheses:

1. $e \in pnat$
2. $x \in pnat\ list$
3. $x_h \in pnat$
4. $x_t \in pnat\ list$
5. $x_r \in (\exists y \in pnat\ list \forall z \in pnat\ member(z, x_t) \supset (z =_{pnat} e \vee member(z, y)))$

   $\wedge \forall w \in pnat\ member(w, y) \supset member(w, x_t)$

   $\wedge \neg member(e, y)$
6. $y \in pnat\ list$
7. $x_{r_y} \in \forall z \in pnat\ member(z, x_t) \supset (z =_{pnat} e \vee member(z, y))$

   $\wedge \forall w \in pnat\ member(w, y) \supset member(w, x_t)$

   $\wedge \neg member(e, y)$

$\Longrightarrow \exists y' \in pnat\ list$

$\forall z \in pnat\ member(z, x_h :: x_t) \supset (z =_{pnat} e \vee member(z, y'))$

$\wedge \forall w \in pnat\ member(w, y') \supset member(w, x_h :: x_t)$

$\wedge \neg member(e, y')$

Correspondingly, a *spread* function is built into the function being constructed:

$$\lambda e \lambda x.list\_ind(x, (nil \& P_{nil}), [x_h, x_t, x_r, spread(x_r, [y, x_{r_y}, \_])])$$

At this point a case-split is made depending on whether x_h is e or not. Accordingly, the result must either skip x_h or include it. The case-split is made

using the refinement $decide(x_h =_{pnat} e)$, which produces two subgoals, each identical to the current proof node apart from a new hypothesis, either $x_h =_{pnat} e$ or $\neg x_h =_{pnat} e$. The construction is developed further:

$$\lambda e \lambda x.list\_ind(x, (nil \& P_{nil}), [x_h, x_t, x_r, spread(x_r, [y, x_{r_y}, pnat\_eq(x_h, e, \_, \_)])])$$

The two gaps will be filled in by the extract terms arising from the subproofs.

These subproofs are about existential formulae. For each, a suitable object is introduced for y, and a subproof verifies that the introduced object meets the specification. The pair of the object and its verification proof's extract term constitutes the extract term.

- If $x_h =_{pnat} e$, x_r is introduced for y. Let the extract term for this be $P_{x_h=e}$.
- If $\neg x_h =_{pnat} e$, $x_h :: x_r$ is introduced for y. Let the extract term for this be $P_{x_h \neq e}$.

The final construction is:

$$\begin{aligned}&\lambda e \lambda x.list\_ind(x, (nil \& P_{nil}),\\ &\qquad [\, x_h, x_t, x_r,\\ &\qquad\ spread(x_r, [y, x_{r_y}, pnat\_eq(x_h, e, x_r \& P_{x_h=e}, x_h :: x_r \& P_{x_h \neq e})])])\end{aligned}$$

The induction hypothesis/recursive call label has been used, so the constructed function is recursive.

Notice that the way in which the induction hypothesis is used determines the way the step case part of the definition relies on the recursive call (to the same function with a reduced value for the recursion argument). Removing the bits representing proofs about the (in)equality of x_h and e, the recursive definition constructed for the step case is essentially $pnat\_eq(x_h, e, x_r, x_h :: x_r)$. The step case of the function is defined to evaluate the condition, and then depending on the outcome of that, yield either just the recursive call itself, or the list constructed by "cons"ing the head of the current list onto the result of the recursive call. This definition may be arbitrarily complicated, and involve other functions which have already been defined.

3.5.2 Defined Induction Schemes

There are constructs for other induction principles (natural numbers and integers) built into the system. More sophisticated ones can be established based on them, such as course of values and induction based on the construction of numbers as products of primes. This is done by proving higher order theorems which justify the new scheme, showing that it is well-founded and complete over the type of objects it refers to. The proof will rely on one of the existing schemes.

The existence of other induction schemes is crucial for programming. The duality of induction and recursion means that the inductions we choose define the recursive structures of the corresponding programs. Many algorithms are distinguished by their ability to access data structures in particular ways and then act recursively on the result, e.g. divide-and-conquer algorithms. Consequently, to achieve such elaborate recursive algorithms, we need an extensive repertoire of induction schemes.

3.6 Synthesis, Verification and Transformation

Correct programs emerge automatically from proofs of their specifications in Oyster. As I have shown in the previous section, depending on how a theorem is formulated, the proof may correspond to a synthesis or a verification or both. Equality proofs only involving universally quantified variables perform no synthesis, only verification. A synthesis proof including a specification will have a verification subproof. To achieve a synthesis of a specified function, f, or satisfying some property P, we need goals of the form

$$\vdash \forall x \exists y.\ y = f(x)$$

and

$$\vdash \forall x \exists y.\ P(y, x)$$

The proofs of these force us to produce a value for y, normally a defined computation on x - this is the synthesis. The proof then continues to verify that

the supplied value for y actually satisfies the equality or whatever property was required.

Transformation of proofs achieves transformation of programs, so it is possible to start with a theorem/specification which would lead straightforwardly to one proof/program, and augment it to get a proof corresponding to an improved algorithm. In chapter 7 I will describe how a theorem corresponding to a naïvely recursive program can be automatically generalised to a theorem corresponding to a tail-recursive algorithm, and proved. The program corresponding to the generalised theorem then has the properties we intended, as enforced by constraints placed on its proof structure. equivalence of the two programs, corresponding to the original theorem, and the generalised one, is assured by the proof too.

3.7 Conclusion

Constructive logic need not be based on Constructive Type Theory, but the consequences of defining it that way are useful. We can use the **proofs-programs** correspondence to extract programs from proofs. Oyster is somewhat restricted by the types available, though this is not too much of a problem for most practical purposes, since most of the ones we use can be simulated or added as new types.

The use of proof in constructive logics is a promising approach to the design of provably correct programs, because the proof structure and program structure correspond and share a common language. The semantics are essentially operational rather than denotational. We are working with the actual program (as a proof). This is in contrast with much other verification and synthesis work which argue about the program as an object. Such approaches have a separate level of describing side effect properties of the specification, such as properties of invariants, and proving them. Constructive logic has the powerful advantage that specification translates integrally and directly into a functional program, modulo a proof.

Chapter 4

Higher Order Unification

As I have previously described, middle out reasoning involves the use of meta-variables to stand for objects which will become instantiated as the proof proceeds. For some cases first-order unification might be enough. However, the meta-variables could correspond to function symbols, in which case second-order unification would be required to instantiate them. In general, middle out reasoning might require even higher-order objects to be represented as meta-variables. An algorithm must be found which can supply suitable unifiers, given terms which may include higher-order variables.

A further requirement of the algorithm chosen is that it must be capable of dealing with the language of the terms it will be given to compare. Unification algorithms are, after all, designed for specific languages, not just different orders of variables. They may or may not accept types, for example. Second-order logic allows us to quantify over functions as variables, and therefore write terms containing functionals. Clearly, whether or not this is a problem may depend on how precise the algorithm is in its language demands and assumptions. Algorithms may be deliberately flexible, so as to avoid such problems, and a little subterfuge may allow terms to pass through algorithms not intended for them.

If two terms in a higher-order language are unifiable, the number of unifiers may be infinite, so algorithms searching for all **unifications** may not terminate. Higher-order **unifiability**, being semi-decidable, is more tractable, as only a process checking the unifiability of non-unifiable terms will fail to terminate.

Huet's classic algorithm for the simply-typed λ-calculus, [Huet 75], is a happy compromise. Although it is a unifiability algorithm, the tree it returns recording its search includes any substitution sets found. Apart from one special case, these substitution sets are complete - they describe all the substitutions which will unify the two terms. The omission is associated with what are termed flexible-flexible pairs of terms. These are similar to variable-variable pairs in first-order systems, but give rise to infinite sets of unifiers, causing non-termination in unification algorithms. Huet's algorithm notes this, and simply records at this point on the search tree that the terms are unifiable, without attempting to supply unifiers.

For use with the various CLAM methods I developed using meta-variables for middle out reasoning, I implemented a higher-order unifiability algorithm based on Huet's algorithm. Interfacing was necessary to handle differences between Oyster's language and the simply-typed λ-calculus. Preprocessing was required to establish types of all objects in advance, and carry out normal-forming. Post-processing, solutions were filtered to respect the quantification in my problems. Adaptations were made to present terms for unifying suitably with respect to quantification, so that universally quantified variables in lemmas were recognised as variables for unification. Further adaptations dealt with the occurs check when it arose, and selected an answer in the case of flexible-flexible pairs. I will describe this in detail in this chapter.

4.1 Huet's Algorithm

The tree of possible unifications is searched and since unifiers, if they exist, will occur at a finite level, they will be found. The algorithm returns its entire unifiability search tree including both these successes and any failures, and selection from the successes is at our discretion.

Huet describes simplifications possible when terms are in η-normal form, which I have incorporated.

4.1.1 Background

Types

The algorithm is designed for a typed lambda calculus consisting of elementary types and the types composed from them using $\rightarrow$, i.e. functions. The theory for the terms over which the algorithm will be used, Martin Löf type theory, has a richer set of type constructors, including set union and allowing dependent types. The algorithm would, therefore, not be validly applied if we used it for arbitrary terms, but provided that we restrict our unifications to terms whose types do not stray outside the simply typed, this is not a problem. We are restricted by this to variables, atomic objects which name constant symbols such as 0 and *reverse* (of any order), terms composed from applying functions to variables and atomic objects, and λ-abstractions of these terms as described below.

It is necessary to know the types of all the elements composing both terms before the algorithm proceeds. This is not decidable for Martin Löf type theory in general, but can normally be managed for most terms we encounter in practice, although sometimes only with ingenuity and persistence.

Terms and Unification

Terms are composed of:

- **Atoms**, which are known constants such as 0, *nil*, *plus*, *reverse* etc. and a countable set of variable names. A finite number of variables may occur in any term to be unified. All of these constants and variables must have known types. Other variable names will be required as the algorithm proceeds, their names are drawn from the set of variable names, and they are assigned types as they are used.

- **Applications**, which are terms like $f(a)$, which has type β when f is of type $\alpha \rightarrow \beta$, and a is of type α. Both f and a are subterms of $f(a)$.

- **Abstractions**, such as $\lambda x.e$, which has type $\alpha \rightarrow \beta$ if e has type β and x is a variable of type α. e is a subterm of $\lambda x.e$, but x is only if it occurs in e.

The notion of a tuple is handled by *currying* in the usual way, so $plus(x, y)$ is treated as an abbreviation for $((plus(x))y)$. In the following the I shall follow Huet in showing functions applied to tuples in brackets in the ordinary way. Their types will be given as $\alpha_1 \times \ldots \times \alpha_n \rightarrow \beta$ where Huet would use $\alpha_1, \ldots, \alpha_n \rightarrow \beta$.

Two terms may be unified if they have the same type and a substitution can be performed of terms of the same type for their free variables in such a way as to make them identical.

Normal Form

The main algorithm requires terms to be in a normal form with respect to the rules of λ-conversion, the type-preserving transitive closure of α-conversion and β-reduction. Huet also describes a simplification applicable when η-normalisation is available too, which was implemented.

Presentation of these rules formally requires some preparatory definitions, which I give here, following Huet closely. I present both an informal and a formal version of the rules. The reader may wish to skip this preparatory material and the formal version initially, and refer back as necessary.

- Context $\mathcal{E}[e]$ denotes a term $\mathcal{E}$ which contains a subterm e.

- Bound An occurrence of a variable x in an expression E is *bound in* E if it occurs in a subterm of E of the form $\lambda x.e$.

- Free Any other variable occurrence is *free in* E.

- Set of Free Variables $\mathcal{F}(E)$ is the set of variables having a free occurrence in E.

- Substitution $E_{\{e/x\}}$ is the term resulting from the substitution of e for every free occurrence of x in E, (e and x must have the same type).

- Replaceability $\mathcal{R}(x,y,E)$ is the property that free x's can be replaced by terms containing y's in E without confusion of bound variables. For x and y of the same type, this is true if and only if all the introduced occurrences of y in $E_{\{y/x\}}$ are free.

  $\mathcal{R}(x,y,E) \Leftrightarrow \forall e.(E = \mathcal{E}[\lambda y.e] \supset$ every occurrence of x in e is bound in $E)$

The rules are:

α This renames the bound variable in an abstraction by a variable which is of the same type, for example from $\lambda x.e$ to $\lambda y.e$, where y must not occur free in e.

Formally: Let $E = \mathcal{E}[\lambda x.e]$. For any $y \notin \mathcal{F}(E)$ such that x and y have the same type, and $\mathcal{R}(x,y,E)$, then $\mathcal{E}[\lambda y.e_{\{y/x\}}]$ follows from E by $\alpha-$conversion.

β This applies abstracted functions to terms of the same type. $\lambda x.e(a)$ becomes $e_{\{a/x\}}$, i.e. e with all free occurrences of x replaced by a's.

Formally: Let $e = \mathcal{E}[\lambda x.e'(E)]$. If $\forall y \in \mathcal{F}(E) : \mathcal{R}(x,y,e')$, then $\mathcal{E}[e'_{\{y/x\}}]$ follows from e by $\beta-$reduction.

η This fills terms out to their full size by adding extra variables of the appropriate type to abbreviated terms lacking their full expression.

Formally: $\mathcal{E}[e] = \mathcal{E}[\lambda x.e(x)]$ where $x \notin \mathcal{F}(e)$.

So if we have a term $\lambda x_1 ... x_n.f(e_1,...,e_p)$ where the type of f is $\tau_1 \times ... \times \tau_q \rightarrow \tau$, with $q > p$ the conversion would be to

$$\lambda x_1 ... x_n, w_{p+1}, ..., w_q.f(e_1,...,e_p,w_{p+1},...,w_q)$$

where the w's are new variables of the appropriate type.

Since in addition I have used the η-converted form of the algorithm, the normalisation used here is that all possible β-reductions are made, and all terms are η-converted. Therefore the resultant normal form is unique modulo α-conversion.

For any expression of the form $\lambda x_1 ... x_n . f(e_1, ..., e_p)$, f is referred to as the *head*, and the e_i are its *arguments*. If the head is a constant or one of the x_i, the expression is *rigid*, otherwise it is *flexible*.

Substitution

A *substitution* is a finite set of pairs, each containing a variable and the expression to be substituted for it, e.g. $\{a_1/x_1, ..., a_p/x_p\}$, where x_i are all distinct variables, each of the same type as the corresponding a_i. The a_i are assumed to be normal-formed.

The result of applying this substitution, σ to some term T, denoted σT, is the normal form of the term $(\lambda x_1 ... \lambda x_p . T)\ (a_1, ..., a_p)$. It makes no difference in which order the substitution pairs are applied.

A substitution is a type-preserving mapping. Since it can only apply to free variables, the head of a rigid term cannot be changed by a substitution.

Two substitutions ρ and σ are composed as follows: $\rho\sigma = \{\rho(\sigma x)/x\}$ such that $\rho(\sigma x) \neq x$ where x ranges over all variables. Since it is defined as a composition of mappings, composition of substitutions is associative. This is essential for the working of the algorithm, which assembles potential unifications as it traverses the terms.

A **unifier** of two terms of the same type is any substitution that makes them equal.

4.1.2 The Algorithm

Overview

The algorithm proceeds by starting at the top of the two term trees, considering the unifiability of corresponding nodes. To start with, we have an empty substitution, i.e. one which indicates no variable bindings. As each pair of corresponding nodes is compared, the current substitution is extended to incorporate any new information. In first-order unification, at each node comparison, there is a single most general unifier to be composed into the unifier assembled so far, if the two nodes are unifiable. In higher-order unification, each node comparison may produce a number of unification extensions which do not subsume each other, all of which may lead to different valid unifiers. Consequently, the algorithm must develop a *match tree* whose arcs are substitutions as the comparison of nodes proceeds. Each node on the match tree corresponds to the current state of comparison of the two terms, starting with the two original terms, but becoming a set of pairs of corresponding subterms (with the unification discovered so far applied) as the comparison proceeds. These sets of pairs of uncompared corresponding subterms are called *disagreement sets.* Each branch of the tree from root to leaf, corresponds either to a unifier (perhaps ending with a record of a potentially infinite set of unifiers resulting from flexible-flexible pairs of terms) or to an attempt at unification which failed.

Example

An example borrowed from Huet's description should illustrate this before I proceed to a detailed description. Suppose we wish to unify the pair $< F(X,a), b >$, where lower case letters denote constants, and upper case letters denote variables. X, a and b are all of type α, and F is of type $\alpha \times \alpha \rightarrow \alpha$.

F is a variable function, so it could be instantiated to anything of appropriate type, that is a function of two arguments of type α returning an object of type α. It must have the normal form $\lambda u \lambda v.E$, where E is a term of type α. Two

operations are used to identify variables in ways which could lead to unifications - *imitation* and *projection.*

- **Imitation.** If E is dominated by a constant, it must be just b, for nothing else could unify with b. The substitution is $\{\lambda u \lambda v.b/F\}$. Applying this substitution gives us a new pair of terms to compare, $< b, b >$, which succeeds immediately, adding nothing more to the unifier.

- **Projection.** Otherwise, E could be a projection onto one of the λ variables in F's normal form, $\lambda u \lambda v.E$. Then that variable may turn into b:

  - u generates the substitution $\{\lambda u \lambda v.u/F\}$. Applying this makes the unification over the pair $< X, b >$. This immediately succeeds, with the additional substitution $\{b/X\}$.

  - v generates the substitution $\{\lambda u \lambda v.v/F\}$ Applying this makes the unification over the pair $< a, b >$. This pair cannot be unified, so this branch of the tree records a failure.

Since normal form is assumed, we know that any other substitution for F would not lead to unification, it would have the wrong type or introduce the wrong head symbol.

Detailed Description of Huet's Algorithm

The algorithm constructs a matching tree for two terms using two main procedures, SIMPL and MATCH, which it applies until a unifier is found or the tree is complete. SIMPL manages the nodes of the tree, MATCH compares pairs of terms.

The algorithm starts with a disagreement set containing just one pair, the original pair of terms, which must have the same type or the unification fails immediately. SIMPL uses this set to construct the first node of the tree. SIMPL operates as follows on a node with disagreement set S:

1. It checks all the pairs in the disagreement set which contain two rigid expressions. Suppose they are: $\lambda x_1^i ... \lambda x_{n_i}^i r_i(e_1^i, ... e_{p_i}^i)$ for $i = 1, 2$. If the headings of any such pair fail to match, either because they have different numbers of $\lambda-$variables[1], or because $r_1 \neq (\lambda x_1^2 ... \lambda x_{n_2}^2 r_2)(x_1^1 ... x_{n_1}^1)$, then no unification is possible using the substitution assembled so far. The node is marked as a failure node, and its branch terminates.

   Otherwise, the pair's members have identical numbers of $\lambda-$variables, and the pair is replaced in the disagreement set by all pairs of corresponding arguments, preceded by the appropriate $\lambda-$variables: $< \lambda x_1^1 ... \lambda x_{n_1}^1 e_j^1, \lambda x_1^2 ... \lambda x_{n_1}^2 e_j^2 >$ for $j = 1 \ldots p_1$. This is repeated until there are no more rigid-rigid pairs.

2. Any pairs which have a rigid element first, then a flexible one, are swopped round, and will be processed by MATCH.

3. If the set is empty or all the remaining pairs are flexible-flexible, unification is possible, and no further substitutions are necessary. The branch terminates, with the node labelled a success. Otherwise the node is returned for further processing, along with its disagreement set.

   Leaving flexible-flexible pairs like this can cause difficulties, because it does not necessarily provide a substitution we can use. At this point the number of solutions is infinite. There are two reasons for this, firstly, even considering the two operations we have used to identify substitutions, there are an infinite number of ways of interleaving them to reach possible matches. Secondly, in order to make the algorithm semi-decidable, Huet omitted three other operations which cause prolific branching, and apply in the flexible-flexible case, those of elimination, iteration and identification.

---

[1]with η-conversion this could only happen if an ill-typed expression were submitted to the unifier, and preprocessing should pick that up before here.

This is the crux of the difference between a unifiability and a unification algorithm, it controls the explosive nature of the search tree, but fails to produce some information.

Now assume that the algorithm is at some node on the match tree generated by SIMPL. The substitution assembled so far is σ, the composition of all the substitutions on the arcs above, and the node is associated with a disagreement set, S, the set of outstanding uncompared subterms of the original two terms under σ. Assuming the node has not already been labelled as terminating with success or failure, a disagreement pair with a rigid element is chosen from the set, and MATCH is used to produce a set of substitution pairs, S', from it. The choice of disagreement pair is significant. By selecting from the disagreement set first-in-first-out, we increase our chances of terminating on failure branches, without affecting performance on success branches. A first-in-first-out strategy forces us to cycle through all the disagreement pairs so we find out if any of them is going to lead to failure in an easily perceived way. A last-in-first-out strategy could let us loop on a single disagreement pair early in the list, when a later one might be due to fail clearly. Finite failure trees are important because they allow us to establish non-unifiability, and hence reject certain choices for identifying our meta-variables.

If S' is empty, the pair of elements cannot be unified, and σ cannot lead to a unification, so the branch terminates as a failure. Otherwise, for each element σ' of S', a new arc is grown labelled by σ' to a new node. The new node is created by applying SIMPL to the disagreement set produced by applying σ' to the current disagreement sets' members.

Lastly I shall describe the MATCH procedure, which takes a flexible term and a rigid term and returns a finite set of their possible substitution pairs.

Since the terms have the same type, and have been η-converted, they must have the same number of $\lambda-$variables. Let the flexible term be $\lambda x_1^F ... \lambda x_n^F F(e_1^F, ... e_{p_F}^F)$, let the rigid term be $\lambda x_1^r ... \lambda x_n^r r(e_1^r, ... e_{p_r}^r)$ and let the type of F be $\alpha_1 \times ... \times \alpha_{p_F} \rightarrow \beta$

The aim is to identify possible matches for F. There are two ways in which the terms could match:

- **Imitation.** For F to imitate r being a constant symbol, it must be:

$$\lambda z_1 ... \lambda z_{p_F} r(g_1(z_1, ..., z_{p_F}), ..., g_{p_r}(z_1, ..., z_{p_F}))$$

  where the g's are distinct new variables of the appropriate type. This substitution is returned. It is the most general one that imitates r, follows the type structure and preserves references to the original binding of F everywhere.

- **Projection.** This covers the case where r is one of the $x_1^r ... x_n^r$. F is projection onto one of its arguments:

$$\lambda z_1 ... \lambda z_{p_F} . z_j(h_1(z_1, ..., z_{p_F}), ..., h_{p_r}(z_1, ..., z_{p_F}))$$

  where $1 \leq j \leq p_F$, and the h's are distinct new variables of the appropriate type. Again, this is the most general form this match could take. All p_F such substitutions are returned.

The set of substitutions is all the substitutions proposed in these two ways.

4.2 Implementation

The algorithm was implemented as described, but some interfacing was necessary to use it for middle-out reasoning with CLAM. The requirements of the algorithm are for two terms in normal form, all the components of which are of known type, and known constant or variable status. Additionally, an indefinite supply of fresh variable names is required.

For flexible-flexible pairs, I generate a single solution, a new variable of the correct type.

4.2.1 Representation of Variables

Middle-out reasoning is implemented in CLAM using Prolog variables, even for higher-order meta-variables. For example, $F(a, X)$ where F and X are variables, is just $F\ of\ a\ of\ X$, since CLAM uses the *of* operator for function application. For higher-order unification, copies are made of the terms with these variables replaced by labels which are Prolog atoms, so that the unifications proposed relate to these labels, and an explicit choice can be made, rather than instantiating one directly. Prolog unification is, of course, only first-order, and would only admit a single solution, whereas we need to explore a tree of solutions.

If any of these unifications result in a partial instantiation of a meta-variable, but introduce new variables as described in the algorithm above, these have to be turned back into Prolog variables on their return to the CLAM part of the system. The advantage of this approach is that a variable may be progressively instantiated by different parts of the proof plan, and that instantiation is then easily available throughout the proof plan, propagated by Prolog's unification. It is amenable to revision at the planning level, if one proposed unification fails.

4.2.2 Interfacing Middle-Out Reasoning and Higher-Order Unification

All meta-variables are treated as variables in the higher-order unification. Additionally, any universally quantified variables in lemmas or hypotheses may take any value we choose, so they are treated as variables too. Anything else is deemed a constant. All terms initially submitted to the algorithm are in η-normal form. so interleaving of steps 4 and 6 below is not required.

As pre-processing:

1. Any outermost term structure not legitimate within the simply-typed $\lambda-$calculus is matched by Prolog unification, as I can safely assume that it only contains constant symbols. This is mainly used to filter out mark-

ers used by CLAM to annotate terms. It is also used when matching an induction hypothesis and conclusion from the sequent:

$$\forall a \forall y \exists z.\ z =_{pnat\ list}\ append(t, a)$$
$$\vdash \forall a \forall y \exists z.\ z =_{pnat\ list}\ append(t, h :: a)$$

Huet's algorithm cannot cope with the polymorphic types of = and the underlying Oyster connectives. For induction hypotheses and conclusions, I can take advantage of the knowledge that their outermost quantification will be identical.

2. Copies are made of both the terms to be compared and their context, which is used for type-guessing. The context of a term to be unified is the sequent or theorem from which it originates. Any Prolog variables in the copies are converted to fresh Prolog constants, but their variable status is noted.

3. The types of the two terms to be compared are tested to ensure they are the same. If not, the algorithm fails immediately.

4. The copies of the terms are normalised by β-reduction.

5. The types of all the symbols in the ground copies of the terms are guessed and recorded.

6. The expressions to be compared are checked to make sure they are in η-normal form.

7. All the arguments throughout the term are checked to ensure that they have the correct types to suit the functions they are in. Although it should be possible to do this as type-guessing proceeds, that is a very tortuous operation, in which types of functors are being guessed from their arguments' types and vice-versa. Consequently I make a second pass, once I have all the information. As this wasn't the main thrust of my work, limited effort was put into refining it, and there is certainly plenty of scope for improvement.

As post-processing, the substitution is applied to the terms, any meta-variable parts becoming Prolog variables again.

4.2.3 Types

The kind of terms matched is restricted to those compatible with the typed λ-calculus. Thus, use of this algorithm is legitimate as it stands, and needs no extension. This restriction is imposed both by choice of terms to submit to the unifying routine, stripping out any CLAM notation, and by some pre-processing, as noted above.

Finding the type of an arbitrary term is not necessarily decidable in Martin-Löf type theory. Although usually possible, it is often very difficult. In practice, though, it is usually feasible. For use with this algorithm, I extended CLAM's usual type guessing mechanism.

I will not spend much time on this, as it is only a tool to supply information to the unifying algorithm.

CLAM has some type guessing built in, so that given an arbitrary instantiated term, it can try to establish its type from information globally available to the underlying Oyster system.

The existing type guessing concentrated on knowing the types of constants and constructors, and establishing the types of functions, and therefore terms occupying their arguments, from definitions. Even then, that would not necessarily establish the type completely, as from a definition we might only be able to find out the types of the recursive argument position, and of the result, but not all other arguments' types. For a variable, a quick check would be done to see if its name appeared as a universally quantified (necessarily typed) variable for the current *Oyster* conclusion - an attempt at using context, somewhat out-of-date for the CLAM work.

I extended this considerably, because it couldn't identify the types of all the variables which are generated during the progress of Huet's algorithm. Further,

it was very sequent-directed, whereas I regularly needed the types of objects from wave rules.

Firstly I gave it context, by using the entire current planning sequent including the hypotheses, or in the case of an expression from a wave rule, the lemma from which it was derived. A great deal of information could be deduced from definitions, quantification, etc.

Secondly, my program assembled the type of any term either top-down or bottom-up, according as information was available. This meant that it would approach any term, $f(a, b(c))$, say, knowing *at most* the type of the whole term, τ_t, and try to deduce the type of the functor alone τ_f, i.e. fill out the blanks in $\_ \times \_ \rightarrow \tau_t$ by looking at definitions. If that worked, it would proceed recursively down to establish the types of the components of f's arguments. If it failed to fully instantiate τ_f this way, it would try the other way round, by finding the types of a and $b(c)$, first.

Standard CLAM retains and uses annotations to align unification in expressions and rules used to rewrite them. Here, those annotations are polymorphic, and so have to be removed because we cannot always give them types which would satisfy the unifying algorithm.

4.3 An Example and Some Problems

Looking at another example shows the algorithm's operation, and some problems that arise. Restrictions on the acceptability of substitutions suggested by the algorithm resolve some of the problems.

Suppose we try to unify $G(a, b)$ and $f(X, Y(c, d))$, where lower case letters denote constants, and upper case letters variables. The constants and variables a, b, c, d, X, Y are of type α, while G and f are of type $\alpha \times \alpha \rightarrow \alpha$. I will assume that both terms have been normalised.

Trying to identify G, projection makes it a or b, neither of which can match $f(X, Y(c,d))$, so these are failure branches. Imitation suggests the substitution $\{\lambda u \lambda v.f(H_1(u,v), H_2(u,v))/G\ \}$. A new arc is grown, labelled by this substitution, to a new node. The SIMPL part of the algorithm labels this node by the disagreement pairs of corresponding arguments, once the substitution has been applied: $< H_1(a,b), X >$ and $< H_2(a,b), Y(c,d) >$ These are both flexible-flexible pairs, so the operation of the algorithm is complete. The suggested substitution is informative about G and X, though perhaps not as much as we would like. It is not clear how we would effect a substitution for $Y(c,d)$, only that we could. To deal with this fully we would need to incorporate the explosive parts of Pietrzykowski's algorithm, or some equivalent. We would need an algorithm to supply us with a stream of the unifiers, from which we could make a selection fitting our purpose.

If, however, we have more information, for example that a and X must unify, we can get more precision from the algorithm. Sometimes, it may be known that certain subterms must unify, and their *prior* unification can then constrain the unification of the whole terms. In fact, with care and knowledge about the overall task this unification is serving, the unification can often be turned into a sequence of matches. That is an immense improvement, as second-order matching is decidable, and matching generally is far more tractable than unification.

In this example, if we unify $G(a,b)$ and $f(a, Y(c,d))$, the algorithm proceeds as before, with the same substitution suggested for G, but the disagreement pairs labelling the node are different. The first one is now $< H_1(a,b), a >$. The algorithm suggests three substitutions for H_1:

- **Imitation.** $\{\ \lambda u \lambda v.a/H_1\ \}$ - This succeeds. Including this part of the substitution into the one we have so far for G, we get

  $$\{\lambda u \lambda v.f(a, H_2(u,v))/G\}$$

  which is valid, but sometimes implausible, since it builds one of its own constant arguments explicitly into G. Such unifiers can usually be rejected.

- **Projection.** $\{ \lambda u \lambda v.u/H_1 \}$ - succeeds, $H_1(a,b)$ is just a.

- **Projection.** $\{ \lambda u \lambda v.v/H_1 \}$ - fails, since $H_1(a,b)$ becomes b.

It is worth amplifying on the solution arising from the imitation operation in this example. The reason why we might wish to reject it is a regular one in these middle-out problems, and results from our knowledge about the use of meta-variables for the problem. We will wish G to be a function whose identity could have been given even before the problem was stated using certain labels, x's and y's for local variables. Further, we do not wish it to be in any way defined by the names invented for variables which appear during proof. This will be described in more detail in 6.3.3. There is effectively a *temporal scoping*[2] which applies to the solutions proposed by the unifying routine. Solutions which build arguments into their identity by $\lambda-$abstraction are acceptable, while those which build in the name of a temporary variable occupying the same location, are not. In the above example, this means that the imitation would be rejected where the first projection was accepted. Note that this only applies to entities which are deemed to have had identities before the start of the problem, i.e. anything not universally quantified within the problem.

The behaviour with flexible-flexible terms is a problem if we wish to partially instantiate a meta-variable using the algorithm and then allow later theorem proving to complete the instantiation. All the infinity of potential solutions available from a flexible-flexible unification should be considered. In order to have some way of continuing with middle-out reasoning, I have taken the solution of noting these flexible-flexible pairs, and using a fresh variable of the correct type as a replacement. This is translated back into a Prolog variable and identified by subsequent middle-out reasoning. This is a valid solution, and, I believe, only postpones a phase of the unifying, taking it up again later when more information becomes available.

---

[2]Dale Miller is to be thanked for this concept.

In practice, it is rarely necessary to partially instantiate. Judiciously applied, most unification results in all meta-variables involved being wholly identified. Usually, by working progressively through the subterms, it is possible to reduce the problem to a sequence of higher-order matches rather than unification.

4.4 Other Algorithms

Other algorithms attempting some form of higher-order unification have already been described in chapter 2. They fall into the following categories:

- They fail to cover all the cases I need. This is the case for F-matching, which will not allow projection.

- They do not terminate when unification is possible, and cause significant search space explosion. I count Pietrzykowski's second-order algorithm, and Jensen and Pietrzykowski's ω-order version here.

- They are developments from Huet's algorithm. Pym's work partially extends Huet's algorithm, but it would require further extension to deal with Oyster terms more generally. The particular form his extension takes is specific to the $\lambda\Pi-$calculus, and would not be suitable for non-Oyster applications of middle-out reasoning.

  Miller's algorithm is also based, at least originally, on Huet's algorithm. He gains control over it by designing it for use on a language in which predicate quantification is impermissable. In effect, it is similar to F-matching. Again, it involves a specialisation which would preclude its being immediately useful.

4.5 Conclusions

Huet's algorithm is a good compromise between efficiency and unmanageable infinite sets of unifiers. The preprocessing and interfacing required is considerable, but unavoidable. Small adjustments handle the cases of flexible-flexible disagreement pairs back to the planner for further information, or select a single valid solution.

By applying problem knowledge, unifications can often be reduced to a sequence of subterm unifications, each of which is actually a match, and therefore becomes decidable. The knowledge about these subterms was available before, and used, via embedded annotations, in standard CLAM. This device was satisfactory for first-order unification, but not for higher-order unification, where explicit separation was required.

Further problem knowledge filters out some unifications which are not valid solutions in a theorem proving context, as they break the rules of temporal scoping.

Chapter 5

Proof Planning

The experiments with meta-variables described in this thesis have been conducted within the CLAM proof planning system [Bundy *et al* 90a, van Harmelen 89] developed by the mathematical reasoning group in Edinburgh. CLAM is described briefly here - for more detail, [van Harmelen 89] should be consulted. Although the results regarding meta-variables are not entirely dependent on this system, the manner of their use is strongly influenced by CLAM's paradigm of theorem-proving, and the search control mechanisms available in it.

Although originally built for use with the Oyster implementation of Martin-Löf Type Theory, CLAM is a separate planning system which can be used to control proof-like operations in various frameworks. These might describe other logics, e.g. [Wiggins 90]. Even if the object-level system is a logic, its proof-objects may or may not have translations into programs *à la* constructive type theory, as described in chapter 3. Although the planner can be viewed as guiding problem solution in general, I will describe it with respect to sequent calculus proofs, since that is the application for which it is mainly used, and for which it is used here.

5.1 The Planning Meta-Level

Reasoning at a meta-level is useful in avoiding a large scale object-level search problem if that search can be conducted effectively amongst a smaller number of meta-level operations. Since our understanding of the structure of proofs is in terms of meta-level properties, it is appropriate for the search to be guided at that level [Bundy *et al* 91b].

In Oyster/CLAM the object level is large in two ways:

- As for any theorem prover, there is a choice of inference to perform. This is the more interesting control problem.

- Additionally, since this is a typed system, there are many, largely trivial, well-formedness goals, which it is normally convenient to ignore at the meta-level.

Although the latter kind of inference can usually be completed automatically without too much search, it would be an undesirable overhead if the well-formedness subproofs were computed for every attempted meta-level proof, since some of these will be false starts which will later be abandoned.

A meta-level may work in different ways. It may be:

- a way of heuristically guiding an object level process by explicit consideration of meta-level domain-related properties at each step. The focus is on the domain, and the meta-level is very tightly defined for the domain in question.

- a description of processes in terms of meta-level properties, so as to be suitable for heuristic guidance at that level, for example in a planning process. The focus here is on broadly defined meta-level control structures. These may be more or less designed to suit different kinds of application and will be tailorable for individual applications. They can include any

object level information, plus additional meta-level information, such as the stage reached in a proof and the path being followed. CLAM falls into this category. In CLAM guidance is through preconditions and following general patterns of proof structure.

- an abstraction, which is a meta-theory in which a set of meta-inference rules apply [Plummer 85, Giunchiglia & Walsh 89, Giunchiglia & Walsh 91]. This is some kind of analogue of the object level. Inference is defined at the meta-level to echo the object level, and search at this level should be cheaper. A resulting meta-proof may then be translated into object level proof. An advantage of this version is the comparative ease of reasoning about its abilities in relationship to the object level.

This list is not intended to be complete or mutually exclusive, just to convey some of the range of possibilities. Within these categories, different choices of meta-level description language will naturally be significant.

In the latter two cases, the problem solution may be completed at the meta-level before any execution at the object level. There may be a hierarchy of meta-levels.

A smaller meta-level may involve less meta-level search, but fail to have the precision to distinguish object level situations well, resulting in considerable object-level search. A larger meta-level may scarcely differ from the object level, and just carry out most of the object-level search at the meta-level.

In addition to degrees of efficiency, two other types of failure are important to consider for any meta-level guidance.

Firstly, how often does the meta-level inhibit the discovery of solutions which exist at the object level? This consideration is more serious for abstractions, since the abstraction is a whole entity functioning in concert, and any change involves a complete re-design. Meta-level planning admits more easily of the alteration and creation of individual meta-level properties and heuristics using them. The totality of the planning heuristics and their use is still an entity whose joint

functioning must be controlled, but it is more flexible. This can be viewed as a trade-off against the loss of some more powerful results such as completeness.

Secondly, how often does the guidance indicate a solution at the meta-level, when there is no corresponding solution at the object level? In practice, this has never happened in CLAM.

The rest of this chapter will look at the CLAM approach to proof plans.

5.2 Main Induction Strategy Proof Plan

5.2.1 Associativity of + Example

Proof plans are best described in terms of an example, such as the associativity of +. Given a definition of +:

$$\begin{array}{rlll} \forall y \in pnat & 0 + y & =_{pnat} & y \\ \forall x \in pnat\ \forall y \in pnat & s(x) + y & =_{pnat} & s(x + y) \end{array}$$

We wish to prove:

$$\forall x \in pnat\ \forall y \in pnat\ \forall z \in pnat\ x + (y + z) =_{pnat} (x + y) + z$$

Introducing the universally quantified x, and performing induction on it produces two subgoals, a base case and a step case:

1. The base case,

   $$\forall y \in pnat\ \forall z \in pnat\ 0 + (y + z) =_{pnat} (0 + y) + z$$

   is easily completed by:

   (a) using the base case of the definition of + twice to evaluate the terms of the form $0 + \ldots$. Evaluation of terms according to function definitions is handled by a **symbolic evaluation** method, described in 5.5.

(b) then the conclusion is just the equality of two identical terms

2. The step case,

$$\forall y \in pnat\ \forall z \in pnat\ x + (y + z) =_{pnat} (x + y) + z$$
$$\vdash \forall y \in pnat\ \forall z \in pnat\ s(x) + (y + z) =_{pnat} (s(x) + y) + z$$

is completed by

(a) three symbolic evaluations using the step case of the definition of plus to rewrite the conclusion:

$$\forall y \in pnat\ \forall z \in pnat\ x + (y + z) =_{pnat} (x + y) + z$$
$$\vdash \forall y \in pnat\ \forall z \in pnat\ s(x + (y + z)) =_{pnat} s((x + y) + z)$$

(b) whereupon the use of the induction hypothesis makes the conclusion another equality of two identical terms.

Provided that a suitable analysis has taken place to choose the induction variable and scheme wisely,

- at least one occurrence of the induction variable will be in an argument position of a function on which the function is recursively defined, so that the definition may be used to symbolically evaluate the terms deriving from this occurrence, and

- the induction scheme will have been selected to create the appropriate values and term structures for the definition to work on, i.e. 0 and $s(...)$ in this case.

Each branch resulting from the induction may need to be rewritten a number of times, by base cases or step cases of definitions, or by using other lemmas already proved. This will be explained in more detail later for the step case. In fact I am describing a simple case here, and later in this chapter, I will describe the more general version.

The overall structure of this proof is an **induction**, after which

- the base case will yield at least one **symbolic evaluation using the base part of the definition** of some function appearing in the conclusion, and

- the step case will yield at least one **rewriting using the step part of the definition** of some function appearing in the conclusion, and should be followed eventually, usually after other rewritings by a **rewrite using the induction hypothesis** - this is termed fertilization, following Boyer and Moore.

We would always expect to make some use of the induction hypothesis in a step case. In general, though, we could not expect either branch to terminate after any particular induction, since further proof perhaps using other inductions may be required to complete it.

This pattern is what we would expect of anything involving an induction with base and step cases. I will examine the key stage more closely in 5.2.2 and show some refinements.

5.2.2 The Key to Successful Induction Proofs

The key to making inductions work is the negotiation of the step case. This is worth examining in some detail.

In the current CLAM system, induction is described mainly using constructors, such as the successor function, 's', or the list constructor, '::'. Other induction schemes based on multiplication or addition, for example, are available. There are innumerable induction schemes, many of exotic construction, but the common ones used in straightforward theorems exemplify the general problem. I will return to the problem of choice of induction scheme later.

Taking c to be some constructor function defining the step of an induction, a typical induction starts with a sequent of the form

H_1

...

H_n

$\vdash P[x]$

where x, a free variable, may occur in various positions in P, as indicated by the square brackets, and $H_1, ..., H_n$ are hypotheses. Depending on the induction scheme, the subgoals will be to prove one or more base and step cases. The latter will be a new sequent (or sequents) of the form:

H_1

...

H_n

$P[x']$

$\vdash P[c(x')]$

To be worthwhile, induction must eventually result in simpler goals to prove. Looking at a step case like this, it appears to be more complicated than its parent. However, when the induction hypothesis is used, the resulting subgoal should be simpler. This therefore, is our aim for the step case of an induction.

In order for the induction hypothesis to be useable, it should appear as a self-contained term in the conclusion, i.e. as $P[x]$, $P[x] \wedge Q$, $P[x] \vee Q$ or the like, so that the induction hypothesis can be used.

Depending on the dominant connective of P, it may be enough to reproduce some subterm of $P[x]$ such that the induction hypothesis enables a rewrite on at least one proof branch. Suppose $P[x]$ has the form $P_l[x] = P_r[x]$, it may be possible to reproduce just $P_l[x]$ or $P_r[x]$, whereupon equality will allow us to rewrite it.

The task then is to take $P[c(x)]$ and apply a collection of rewrites such that it turns into $c'[P[x]]$. Even if some subterm of $P[x]$ is to be produced, the same kind of process applies. We need to rewrite, using lemmas or definitions, to *ripple* the "difference" between the two terms, initially "$c(...)$" upwards towards the root of the term structure, making sure as we proceed that the pre-induction term structure is being preserved for later fertilization. This "difference" is termed a *wavefront*, and tracked as the proof moves it through the term. It will be distinguished here by a surrounding box, and underlining the inner term (the *wave hole*) which is to be preserved. The arrow signifies the direction in which we are expecting the obstruction to move in the term structure, more explanation of this will be provided later. The simplest such rewrites, termed *wave rules* because they have this rippling effect, have the form:

$$f(\boxed{c(\underline{x})}^{\uparrow}, \vec{y}) \Rightarrow \boxed{c'(\underline{f(x, \vec{y})})}^{\uparrow}$$

E.g. the definition of $*$ (multiplication):

$$\boxed{s(\underline{x})}^{\uparrow} * y \Rightarrow \boxed{\underline{(x * y)} + y}^{\uparrow}$$

More general forms take account of multiple wavefronts, nested functions and multiple variables within the wavefront. These are described in [Bundy *et al* 90b, Bundy *et al* 91b, Bundy *et al* 91a].

By marking this wavefront and tracking its progress, we can ensure that our rewriting is directed towards the desired effect, e.g. raising the wavefront in the term structure so that enough of the induction hypothesis is restored to permit a rewrite. To achieve this may take a number of co-ordinated ripples. As we shall see later, there are other means of removing obstructive wavefronts.

Step cases of recursive definitions have exactly the right form to achieve this effect, but existing theorems may too, perhaps in more than one way. For example the associativity of *append*:

$$append(x, \boxed{append(\underline{y}, z)}^{\uparrow}) \Rightarrow \boxed{append(\underline{append(x, y)}, z)}^{\uparrow}$$

$$append(\boxed{append(x, \underline{y})}^{\uparrow}, z) \Rightarrow \boxed{append(x, \underline{append(y, z)})}^{\uparrow}$$

yields two wave rules derived from a single theorem. This effect is designated as *longitudinal*, because it raises the wavefront in the term.

Not all definitions move wavefronts upwards, some of them move them sideways, or *transversely*:

$$f(\boxed{c(\underline{x})}^{\uparrow}, \vec{y}, a) \Rightarrow f(x, \vec{y}, \boxed{\underline{a}}^{\downarrow})$$

Both forms of wave rule mark the movement of a portion of term structure around a constant skeleton term. In both cases, the direction of movement is important. In the longitudinal case, the upward movement leaves behind increasingly larger contiguous pieces of skeleton. This leads towards the possibility of reproducing sufficiently large skeleton subterms that fertilisation can take place. In the transverse case, the portion of term structure being moved remains within the skeleton, but moves sideways towards a variable which is still universally quantified. Here, too, fertilisation can take place, because that variable can "soak up" the extra term structure, as the hypothesis copy of the variable need not be instantiated to the same value as the conclusion copy.

An example of a transverse wave rule is the tail-recursive definition of *reverse*:

$$reverse(\boxed{h :: \underline{t}}^{\uparrow}, a) \Rightarrow reverse(t, \boxed{h :: \underline{a}}^{\downarrow})$$

Notice that the wave hole notation on the right still indicates the part of the expression which persists from before the application of the rule. Lemmas may also move the wavefront transversely:

$$append(x, \boxed{append(y, \underline{z})}^{\uparrow}) \Rightarrow append(\boxed{append(\underline{x}, y)}^{\downarrow}, z)$$

$$append(\boxed{append(\underline{x}, y)}^{\uparrow}, z) \Rightarrow append(x, \boxed{append(y, \underline{z})}^{\downarrow})$$

Here the depth of the wavefront in the term is not changed, but the direction of the arrow has.

This notion of moving a wavefront transversely is useful when both the induction hypothesis and the conclusion retain some universally quantified variables, as in:

$$\forall y_h.\ P[x', y_h]$$

$$\vdash \forall y_c.\ P[\boxed{c(\underline{x'})}^{\uparrow}, \lfloor y_c \rfloor]$$

This new piece of notation, $\lfloor \ldots \rfloor$ focuses on y_c, annotating the location in the conclusion which corresponds to a universally quantified variables in the hypothesis. Such variables may be instantiated to any value. They are called *sinks*, because wavefronts can be "poured" into them. Indeed, CLAM detects when this is happening, and expands the sink annotation to include the captured wavefront.

If there is a rewriting which can move the wavefront to be around y_c:

$$\forall y_h.\ P[x', y_h]$$

$$\vdash \forall y_c.\ P[x', \boxed{\lfloor c''(\underline{y_c}) \rfloor}^{\downarrow}]$$

then we can give y_h a value to suit, i.e. $c''(y_c)$. This kind of operation is useful in proofs involving variables designated as accumulators, the rôle played by the y's here, since it can have exactly the desired effect of moving a wavefront onto a sink acting as an accumulator. The notations to mark sinks and distinguish the directions of wavefronts are recent additions to CLAM, and were not available when I was working on it for tail-recursion optimisation. Consequently I will use the notation relevant to the version of the CLAM system available to the work being described. I shall return to the use of accumulators later, in Chapter 7.

Characterising the effect of different usages of lemmas and definitions on key components of the sequent is essential to defining the structure of proofs. The terminology describing this forms part of the meta-language used for planning. This approach has the advantage that different formulations of the same concept can be accommodated naturally. Since the different formulations are provable from each other, the theorems stating their relationship become available lemmas classified by their rewriting effects, as in the Boyer-Moore theorem prover. This is as true for base case rules as it is for wave rules, so the symbolic evaluation method including the defined base case $append(nil, x) \Rightarrow x$, would also include $append(x, nil) \Rightarrow x$ as a reduction rule, once proved.

5.2.3 Choice of Induction

The choice of induction scheme is undertaken by the induction method, and also incorporated into the overall induction strategy.

The initial choice of induction is crucial to successful management of step cases, since it will define the initial sequent. It involves two choices, of the induction variable and of the induction scheme. Induction must take place on a free variable, so any variables which are free at that point in the proof, or can be made so because they are universally quantified in the conclusion, are candidates.

If the selected variable is universally quantified, it is made free by an introduction rule of inference (Oyster proofs work backwards, remember, so introduction rules rather perversely seem to remove connectives - they are of course introducing them in the forwards direction). This will make the first universally quantified variable free, so CLAM does a little work to re-arrange leading universal quantifiers so that only the required one becomes free. As we have seen, it may be advantageous later to have the others still quantified, as this will make the induction hypothesis more powerful.

Following Boyer and Moore's induction analysis, CLAM's induction selection is guided as follows:

1. Find a variable, x, say, on which induction could be performed,

2. Check that at least one of its occurrences is in an argument of a function such that a wave rule applies,

3. Take an induction scheme from the list of schemes CLAM knows, and check that for all the occurrences of x in positions where wave rules are available, the scheme will create a term such as $c(x)$ at that position such that a wave rule applies,

4. Preferably, but not necessarily, all occurrences of the variable will have some wave rule available.

If any of these steps fails, it backtracks and tries again.

This insistence on availability of wave rules initially sets up something with a good chance of enabling a successful ripple to fertilization of the step case. If that stage fails, within the attempt to apply the whole induction strategy, backtracking will try other inductions on this sequent.

5.2.4 Induction Strategy

The general structure already suggested can now be refined further. I will refer to this below as the "Induction Strategy", when I explain how it is incorporated into the planner. We expect the selected induction to lead to the following two kinds of subgoals.

Base Case

The free variable selected for induction will have been replaced throughout by a constant. Simplifications by rewriting using base cases of definitions may apply, and also similar results which have been proved as theorems, involving base case values such as 0 and *nil*. Achieving some simplification like this may enable others, so *symbolic evaluation* , which is used for base cases, is allowed to include any *reductions* processing a constructor symbol. For example, rewriting $(0 + s(y)) + z$ to $s(y) + z$ makes it possible to continue rewriting using the step case of $+$ where that wasn't possible before. Rewriting using equalities present as hypotheses is also tried. It is quite difficult to draw a hard line between symbolic evaluation, and other kinds of rewriting, and the constituents of symbolic evaluation are revised from time to time in CLAM.

Step Case

This has already been described. It consists of rewriting by step cases of definitions, or other longitudinal wave rules, or failing them transverse wave rules, until simplification takes place by fertilization.

5.3 Constructing Proof Plans

The construction of the plans follows the shape of the proofs, which are tree-shaped. At each stage, the planner has a partial proof tree, and chooses a leaf to work on (see section 5.6). The leaf will be characterised by the sequent derived at that point in the meta-level representation of the proof (I will refer to this as a *meta-proof*). The action of the planner will be to complete that portion of the meta-proof, or find a way of developing it, which will extend the tree to new leaves, and then recurse. When there are no more leaves to work on, the proof is complete, and the plan construction has succeeded.

Proofs are analysed to establish regular components. *Methods* are designed corresponding to these components, so that the planner's linking of them corresponds to building a meta-level representation of the proof. Each method is designed to detect whether the circumstances in which it is applicable pertain, and then compute a *tactic* - an appropriate collection of object level inferences, tailored to have the desired effect on the particular sequent. Methods are described in detail in section 5.4. The particular combination of methods which forms a meta-proof is also that which combines all the methods' tactics to form a proof.

CLAM uses the Induction Strategy plan, if possible, to save it from considering all the methods individually. Commonly, a number of applications of this will complete a meta-proof, perhaps with some symbolic evaluation to tidy up.

In the event of the Induction Strategy failing to lead to a solution, the planner will try to apply the components separately, since a more flexible combination may succeed.

This design makes CLAM adaptable to new problem areas since it is easy to incorporate new purpose-built methods.

Since CLAM is built in Prolog, Prolog can easily be used for arbitrary computation within it, as required by the methods.

5.4 The Plan Components – Methods

5.4.1 The Description of a Method

Methods are described by a frame structure as shown in figure 5–1.

method:

name(...Args...)	name slot: Prolog term
H ⊢ G,	input slot: sequent
[...Preconditions...],	preconditions-slot: list of conjuncts
[...Postconditions...],	postconditions-slot: list of conjuncts
[...Outputs...],	output slot: list of sequents
tactic(...Args...)	tactic slot: Prolog term

Figure 5–1: The General Form of a Method.

Each method defines a mapping from an input sequent to a list of output sequents. The mapping may be partial, since it may not be fully instantiated. The slot usage in these frames is now described.

- **Name** The method's name and some arguments which will parameterise it in use. Some arguments may be used as output, to communicate with other methods in a combined plan.

- **Input** This becomes instantiated to the meta-level representation of the sequent[1] the method is to work on. Since the frame is stored as a Prolog term, individual methods may require a certain structure of the input sequent, thus imposing implicit preconditions.

---

[1]If the sequent representation currently contains meta-variables, it is not strictly a sequent.

- **Preconditions** These are explicit preconditions which are tested to decide whether the method is applicable. Although they could be arbitrary pieces of Prolog code, it is conventional to use a specially designed library of predicates which describe just those qualities that are interesting in a sequent, such as subterm structures, recursive structures, results of substitutions etc. This description language is also used for postconditions, so there is some uniformity, and a common way of describing the results of the successful application of the method. The connectives for this language incorporate those concepts which are appropriate to the meta-language of theorem-proving, such as "forall", "not" and "thereis" to encourage planning in those terms.

  An important feature of this method description language is the ability to use other methods on a given sequent and discover what the effect of applying them would be. This is used to create compound methods, as described below.

  The execution of the Prolog clauses in both this slot and the Postcondition slot may cause Prolog variables shared with other parts of the frame, such as the Output, to be instantiated. Backtracking may occur.

  There is no guarantee that the method will succeed if its preconditions are satisfied, but they are some protection against attempts which will fail.

- **Postcondition** If the preconditions succeed, the postconditions will succeed too. These are explicit postconditions which will be true after the method has successfully applied to the sequent. The postconditions perform further computations to generate the subgoal sequents.

- **Output** This is a list of sequents which are the subgoals remaining to be proved after the method has been applied to the input sequent. For methods which complete a proof branch, this list will be empty. Since this is the information which goes forward to the planner's continuing attempts at proof, it is important that it be as detailed as possible.

- **Tactic** This Prolog term is the actual tactic which will be used on the object level sequent. It is a macro of inference rule applications, or the name of one.

5.4.2 Example – The Wave Method:

This is the definition of the method which guides the use of longitudinal wave rules.

```
method(wave(Pos,[Rule,Dir]),
       H==>G,
       [matrix(Vars,Matrix,G),
        wave_rule(Rule,long(Dir),L:=>R),
        exp_at(Matrix,Pos,L)
       ],
       [replace(Pos,R,Matrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG)],
       [H==>NewG],
       wave(Pos,[Rule,Dir])
      ).
```

The arguments to the name (`Pos,[Rule,Dir]`) may already be instantiated at the time of use, or may become instantiated through use. Oyster and CLAM use `==>` for $\vdash$. `H==>G` will be instantiated to the input sequent.

The list of preconditions operates as follows:

- `matrix(Vars,Matrix,G)` requires that `G` is a formula `Matrix` universally quantified by `Vars`.

- `wave_rule(Rule,long(Dir),L:=>R)` instantiates the variables to respectively: the name of a wave rule, its direction of use, the form of the expression which may be rewritten using the rule and the form of the result of

the rewriting. Wave rules are identified as a pre-processing step when all the definitions and lemmas are loaded. They are stored with Prolog variables for the universally quantified variables for ease of checking matches. Backtracking over this will produce other wave rules.

- `exp_at(Matrix,Pos,L)` requires that subexpression, `L`, of `Matrix` is at position `Pos`. By Prolog unification it is constrained to match the left hand side of the wave rule. By backtracking, all such subexpressions can be found, depending on the success of later conditions.

Having identified a candidate wave rule rewrite, the Postconditions calculate the result:

- `replace(Pos,R,Matrix,NewMatrix)` means that `NewMatrix` and `Matrix` are identical, except at position `Pos`, where `NewMatrix` contains `R`. I.e. `NewMatrix` is `Matrix` with the rewrite applied.

- `matrix(Vars,NewMatrix,NewG)]` `NewG` is `NewMatrix` with the universal quantification restored.

The single output sequent is then constructed from the original hypotheses and the new conclusion formula. The actual tactic is suitably parameterised by the identification of `Pos` and `[Rule,Dir]`.

5.4.3 Compound Methods

The major methods which describe common proof structures may be created by combinations using analogs of the tacticals (derived from LCF) used to combine tactics:

- Method1 *or* Method2 - attempts Method1, and then Method2 as an alternative if Method1 fails.

- *try* Method - attempts Method, but does not fail (as a precondition for example), if Method does. This can be useful if a particular method is

likely to apply. If it does, progress has been made, but if not, we don't want its failure to invalidate the applicability of the current method. If it fails, it passes on as output sequent just the input sequent it received.

- Method1 *then* Method2 - attempts Method1 and then on each of the subgoals so produced, Method2. A variant syntax which is often useful is Method1 *then* [Method2a, ..., Method2n] where the list of methods is attempted on the subgoals. The first of the list applying to the first subgoal, the second to the second etc.

Repetition is achieved by creating new methods which are defined just to be the iteration of others. An *iterator* construct takes a list of methods and a representation of a sequent, and builds a new method which will apply the methods from the list exhaustively. Such constructs are added when the methods are loaded up initially.

In addition to the methods, there are submethods, which are defined in the same way as methods, but which are not available to the planner independently for plan formation, and may only be used as components of other methods. This facilitates the organisation of logically discrete entities which are not used at the planning level.

The general strategy described in section 5.2 starting with an induction, and proceeding to fertilization with the induction hypothesis, is built from the methods described in the next section, linked by the combinators above.

5.5 The Basic System

The basic CLAM starts with some methods supplied, which the user may add to or remove. They are:

- **tautology** - for identities of the form $X = X$, where X may be universally quantified, and simple logical tautologies. Like symbolic evaluation, this is subject to periodic revision.

- **sym_eval** - the symbolic evaluation method, is an iterator construction of the following:
  - **equal** - rewrites of variables using equalities of the form $variable = term$ or $term = variable$
  - **base** - rewrites using base cases of definitions.
  - **step** - rewrites using step cases of definitions.
  - **reduction** - rewrites using lemmas corresponding to alternative definitions, involving constructor symbols such as $0, nil, s, ::$.
- **wave** - looks for the applicability of a wave rule, preferring longitudinal to transverse rules.
- **casesplit** - if there is a conditional wave rule that might apply and the condition can be identified as one of a complementary set, then the case-split corresponding to the complementary set can be introduced.
- **strong_fertilize** - when the current goal has been rewritten so that it matches the induction hypothesis.
- **weak_fertilize** - when all possible rewriting has applied to the goal, but it doesn't match the complete induction hypothesis, sometimes the induction hypothesis, can still be used to rewrite a subterm of the goal. We call this weak fertilization. This is possible if it is an equality, implication or dominated by another transitive predicate.
- **generalise** - generalisation of repeated non-trivial terms, as performed by the Boyer-Moore theorem prover.
- **existential** - takes a goal of the form $\exists x.P(x)$, and generates the subgoal $P(X)$, with X a (Prolog) meta-variable, and associated tactic $intro(X)$. This is the form of the eventual subgoal, and with luck, subsequent proof will instantiate X. In fact this is a limited version of MOR.

- **ind_strat_I** - the basic induction strategy described earlier.

- **induction** - chooses an induction by finding an eligible variable and suitable induction scheme. The variable must be free or universally quantified in the current goal. It must occur in a subterm such that the function surrounding the variable is recursively defined on the argument position it occupies. There must be a suitable induction scheme (these are all defined externally) which subsumes all the schemes suggested at all the occurrences of the variable. If possible, there should be no occurrences of the variable in non-recursive argument positions of functions, this corresponds to Boyer and Moore's *unflawedness*.

  The subgoals produced are the base and step cases corresponding to the induction scheme.

5.6 Search Strategy for Plan Formation

The planner may only combine methods by assessing their applicability at each sequent and constructing a tree of applications of corresponding tactics. It starts with the current sequent, and tests to see which of the methods can be applied, in a user-determined order. Methods which cause termination of a proof branch, such as symbolic evaluation and strong fertilization are tested first. After that, it is sensible to prefer methods which are cheap, in the sense that they reduce, or at least do not increase the complexity of the sequent to be proved. Expensive methods like induction and generalisation, which increase complexity, come at the end of the list.

Development of the plan depends on the search strategy in use:

- Depth-first. Takes the current sequent and extends it as indicated by the first applicable method. The next current sequent is the first of the child sequents.

- Breadth-first. Takes all applicable methods and branches the search tree for the effects of each of them. On each branch, the new current sequent is the first child sequent.

- Iterative-deepening. Similar to breadth-first, but cheaper. The search space is explored to a finite level each time, developed by depth-first search down each branch.

- Heuristic. Heuristic control over the selection of methods.

Overt control over the choice of sequent to develop, can only be achieved by tweaking the construction of the list of output sequents from each method. It is difficult to achieve this through the heuristic planner, since at that level, detailed strategic information is not readily available. This is a weakness of the system which should be tackled, especially for using meta-variables.

Further control is achieved by combining methods into a strategic plan. Emphasis is thus placed on completing proof segments which match that structure, i.e. including a fertilization in step cases. This makes it easier to gauge when an induction choice has succeeded, because the components are explicitly linked as part of a compound method. If left separate, on failing to reach a fertilization, the system could develop a badly chosen proof branch even further by more inductions, when it should retrace its steps and reconsider the previous induction.

5.7 Library

CLAM is designed to be a planning shell for theorem proving. It has a considerable library of definitions and theorems available which may be loaded and classified by their effect. Users' own theorems and definitions extend this. Methods are just other library objects, which may likewise be borrowed or adapted. This makes it very easy to experiment with new classes of proof.

5.8 Comparison with the Boyer-Moore Theorem Prover

CLAM is an effective and flexible tool, proving many theorems from the Boyer and Moore corpus. Those on which it fails mainly involve differences in the type system which makes some of the functions difficult to define when they are required to be total, or are in domains we have not explored yet.

It improves on the Boyer and Moore theorem prover, in the sense that it can structure the linking of its components at an articulated meta-level, subject to the explicit control of a planner.

Chapter 6

Middle-Out Reasoning

In this chapter I shall discuss middle-out reasoning in general, explaining why it is needed, addressing its potential uses and describing some control issues. Subsequent chapters detail my development of middle-out reasoning solutions to particular classes of problem.

Many proofs can be made routine to some extent. There are decision procedures for propositional logic, both classical and constructive. Certain classes of theories in first-order predicate calculus are decidable. For example if a theory has no function symbols, and only monadic predicates, there is an essential finiteness about it, since its Herbrand Universe must be finite. This curtails a source of explosiveness in applying inference rules, in that quantified variables can only refer to a finite number of objects. For such theories there are algorithms which are guaranteed to find a proof if one is available, and to terminate if no proof is available. They will co-ordinate the application of inference rules, and may even choose suitable objects for existentially and universally quantified variables[Wallen 90].

Even within a decision procedure, there may still be choices to be made. Attention can be further directed towards controlling search, so that proofs are found efficiently, and they are "good" proofs according to some criterion such as economy.

In general, though, there are certain kinds of steps which are inherently **explosive**, or certain domains make them explosive, and the procedures above

work precisely because they don't have to deal with such problems. These problematic steps are associated with certain inference rules or with axioms. For most interesting theories these steps are unavoidable:

- Introducing objects for existentially quantified variables in a goal. Eliminating universally quantified variables in a hypothesis. If functions are available as part of the domain, there are clearly an infinite number of objects which could parameterise the inference rule.

- Induction - there may be an infinite number of schemes.

- Using the cut rule to sequence in a new hypothesis which may then be used to justify the current conclusion, but must also be proved in its own right.

There are two aspects to this problem of using inference rules which lead to an explosion of the search space. Firstly we must decide when to use these explosive rules - at each of the many nodes of the proof tree at least one such inferences is available, but may be superfluous, and could complicate the proof needlessly. Secondly we must choose how to parameterise them - these steps involve new terms. Unfortunately, it will not normally be obvious what the parameterising term should be. This is because we need to introduce a term or formula which will be required at a later point of the proof, and which will only become identified at that point. A way of finessing part of this difficulty is to work on the proof at the meta-level, where the rigour of the inference rules may be postponed.

This seems to correspond to human proofs, where there appears to be a grasp of the overall proof structure which enables us to hypothesise a sketch of the proof and then fill in the gaps. In general, the sketch may involve arbitrary amounts of detail, perhaps just a pattern of applications of certain types of inference rule, maybe including some indication of the way the terms or variables are to be handled. Whatever the degree of detail is, such a structure must be selected, adapted and applied. Without one, it is very difficult to control such processes in general.

6.1 When to Apply Explosive Steps

The question of choosing *when* (including *whether*) to apply the explosive steps has not been dealt with extensively in this thesis. Some straightforward options are available which make it possible to concentrate on the parameterisation choices, which are more tractable for an initial study.

To choose when to apply explosive steps middle-out would require an explicit representation of the proof with variable parts of the structure becoming instantiated to methods or proof steps. That would involve higher-order unification on a large scale and would be difficult to control. The alternative I have chosen is to use CLAM's planning system, which constructs plans flexibly but has an implicit representation of the proof.

Development within the CLAM system allows proofs to be planned using "methods" to assess the applicability of macros of inference rules. The plan corresponds to a whole proof, so the planner's selection of methods corresponds to a choice about when to attempt particular kinds of inference. To experiment with MOR, I wrote new methods and adapted some standard CLAM methods. The methods which initiate some of the problematic structures were given very low priority, to prevent the explosive choices occurring frequently, and designed only to apply in limited circumstances. These factors controlled the selection of these methods by the depth-first and iterative-deepening planners, which have been described in chapter 5.

By leaving some of the details as meta-variables, a whole class of plans, corresponding to all the possible instantiations of the meta-variables, can be described in one move. This will be explained in the next section.

6.2 How to Apply Explosive Steps

The general idea here is that we know roughly what we're going to do but not some of the object level details yet. It is possible to exploit the planning meta-level to construct a plan using meta-variables, instantiate them while planning, and then execute the instantiated plan. The basic CLAM system already does a little MOR for existential goals, which is described below. The work of this thesis was to take this approach, and extend it to other kinds of proof.

For some of these kinds of proof the instantiation of the object might happen progressively over some stages of method application. Sometimes identifying the object(s) concerned requires higher-order unification.

I concentrated on certain classes of proofs involving generalisation. Particular types of proof were chosen and analysed, so that their structures would be known and could be used to build methods. These new classes of proof were then added to the existing repertoire.

6.2.1 Existential Goals

Suppose we have a goal:

$$Hypotheses \vdash \exists x \in t.P(x)$$

In principle, any validly typed term constructed from the Herbrand Universe and free variables in the sequent might be introduced for x. Often though, some simple MOR can choose a suitable object to introduce.

This is the only piece of MOR the unadapted version of CLAM does already. At the method level, a meta-variable, Y, ranging over terms of type t is used for x. The meta-level proof process can proceed as if the appropriate inference rule had been applied which would normally require a named object introduced for x. Sometimes this will instantiate Y suitably.

For example in:

$$a \in t \vdash \exists x \in t.\ x =_t a$$

The next sequent in the method search space will be:

$$a \in t \vdash Y =_t a$$

and one of the basic methods (called tautology, but used for various purposes) immediately unifies this with the reflexiveness of equality (the axiom is stored with Prolog variables, as $E = E$), instantiating Y to a. This in turn instantiates the specification of the existential introduction.

Although similar approaches could apply for universally quantified hypotheses, most of our proof plans are asymmetric. They tend to concentrate on the conclusion of the current sequent. Instantiation of universally quantified variables in hypotheses is achieved by methods which check explicitly whether any of the hypotheses could support the current conclusion, if necessary by suitable instantiation. Only then is a universally quantified hypothesis instantiated.

As it stands, the existential method is too eager. It will be satisfied with the first term found to complete a proof branch. If we want the proof to proceed with some less easily found term standing as existential witness, we must restrain the existential method or force backtracking to occur.

6.2.2 Induction

Induction requires us to reason both about a scheme that will work for the problem at hand, and its inductive structure.

For $\mathcal{MOR}$ to select an appropriate induction scheme for x, in

$$\vdash \forall x \forall y.(even(x) \wedge even(y)) \supset even(x + y)$$

we could postulate a step case of the form:

1. $x' \in pnat$

2. $\forall y.(even(x') \wedge even(y)) \supset even(x' + y)$

$\vdash \forall y.(even(C(x')) \wedge even(y)) \supset even(C(x') + y)$

where C is a variable function representing the step case construction. The action of the planner should then be to instantiate C.

Difficulty arises here, not just in this instantiation, but in the indeterminate nature of the inductive subgoal structure it implies. CLAM's methods are quite self-contained, and each one has information about the subgoals it produces. The subgoals from the basic induction scheme over the natural numbers are one base case and one step case:

$$\frac{P(0) \qquad \forall x' \in pnat.P(x') \supset P(s(x'))}{\forall x \in pnat.P(x)}$$

A scheme whose step case skips by n applications of s (denoted by s^n) has n base cases and a step case:

$$\frac{P(0) \quad \ldots \quad P(s^{n-1}(0)) \qquad \forall x' \in pnat.P(x') \supset P(s^n(x'))}{\forall x \in pnat.P(x)}$$

The induction method would not be able to generate the appropriate list of subgoals until C had been instantiated. It would have to create a variable list of subgoals. Once the proof structure becomes variable like this we are engaging in middle-out planning rather than MOR. For arbitrary induction schemes, there may be a variety of base and step cases.

6.2.3 Using the Cut Rule

This is the mechanism that allows us to add any new hypothesis we like to the proof. Clearly this has great potential for explosive proof steps.

As described in chapter 3 this is known as the *seq* refinement in Oyster. Two subgoals arise:

- To prove the original conclusion given the new hypothesis as well as the original ones,

- To prove the new hypothesis from the original hypotheses.

By analysing the purposes of applying this rule, some control can be regained. Typically, using the cut rule falls into one of the categories described below.

Generalisation

The new formula is a generalisation of the current conclusion. This covers useful and challenging classes of problems. There are four tasks - deciding to generalise, choosing a generalisation, proving it, and using it to justify the original formula, i.e. showing that it *is* a generalisation.

MOR is particularly useful for generalisation, as a generalisation involves the introduction of a new formula, and we need to have a way of choosing it. As a result of my analysis of generalisation (chapter 9), I believe I have shed some light on this. My analysis provides a much needed structure, both for assisting MOR, and to inhibit over-generalisation. Since the whole of chapter 9 is devoted to this, and a variety of examples has already been provided in chapter 2, I shall not go into it any further here.

Case Analyses

The new formula establishes a case split. Common examples are:

- $(x < y) \vee \neg(x < y)$
- $(x = y) \vee \neg(x = y)$

assuming any two natural numbers, x and y.

Although the law of the excluded middle is not generally valid in constructive logic, it is alright in those instances where a procedure can be given which will test which of the disjuncts is the case, given arbitrary inputs of the correct types. The proof obligation for such a formula consists of the definition of a decision procedure.

Although this kind of task involves MOR, to decide the formulae involved in the case-split, the meta-level problem of deciding *when* to introduce a case analysis lies more in the realm of middle-out planning than MOR. The reason for this is that case splits change the overall proof structure more than generalisations do, and detecting the need for a case split may require knowledge about the state of more than one proof branch.

Let's consider how we know a case split is needed. Each case of the split normally corresponds to a condition. Each condition is an assumption needed to complete one branch of the proof. If we are only looking at one branch at a time, it is difficult to distinguish an arbitrary subgoal from a component of a case split we should have introduced several stages earlier. There may be clues, perhaps if we use a definition defined in cases, even then, the exact form of the split may not be obvious. A clear sign of needing a case split is when the set of outstanding subgoals for some sub-proof-tree correspond to a known set of mutually exclusive disjunctive cases. So the discovery of the need for a case split usually happens in an otherwise successful proof, and the existing steps can still be used.

The effect on the proof-tree of introducing the case split is also different from generalisation. We take the existing proof, and just inserting the extra assumptions makes it work, or at least the branch we have been proving so far works. The proof of the validity of the case split need have no similarity to the main proof. Explicit proof plan manipulation is required, neither the point at which the case split will be required, nor the kind of case split it should be, can necessarily be determined at the proof node where the split should be inserted.

This is quite a different relationship to the original proof structure from generalisation proofs. In them what was previously the main proof becomes fairly trivial. The proof of the inserted generalisation is similar, but different to the failed proof attempt. Although generalisation involves the planning level too, we backtrack and start planning a different proof rather than an insertion into the existing proof.

The ease with which we can detect and carry out a case split will depend on the planning system in operation, more than just the reasoning system.

Using Lemmas

The cut introduces a lemma which allows us to rewrite some expression in the sequent, i.e. to perform a substitution. Sometimes, the current sequent will just be a special case of the lemma - this is already dealt with by CLAM's standard methods.

6.3 $\mathcal{MOR}$ and Unification

The meta-variables involved in $\mathcal{MOR}$ may be first-order or higher-order. Consequently, higher-order unification is used to identify the meta-variables.

Even if a variable is first-order, we cannot necessarily use just first-order unification to identify it, because it might appear within a term couched in language which is not first-order. This can occur if function variable symbols are present, even if, when $\beta-$reduced, the expression involves no function variables. An artificial example might be unifying $X + 0$ and $((\lambda u \lambda v.v)(*)(+))(s(0))(0)$ where X is a variable. $\lambda-$conversion should take care of this kind of problem, but it would not note scoping due to universal quantification, as in $\forall f.\ \Sigma f(s(x)) = f(s(x)) + \Sigma f(x)$ we would have to be careful not to confuse these f's with others.

So using unification algorithms which are higher-order than we need covers all the sorts of variables we might need, in that it will handle n^{th}-order variables for any n. We could consider reverting to first-order unification when only first-order variables occur in the term, even if it does strictly contain entities which look like constants but are actually second-order variables, such as f above. It is debatable whether it would be worth the effort.

6.3.1 Types

Working in a typed logic, MOR must also establish the types of any new meta-variables. These types are required for the higher-order unification algorithm and they are apparent from the context. We are always in the position of inserting meta-variables to fulfill particular rôles. Sandwiching them into existing expressions tells us what their types must be, if we can guess the types of the components of the existing expression. Although not decidable in general, these can be guessed in practice for all our theorems.

6.3.2 Using Embedded Control Notation

Some of the information available to us about wavefronts, for proof control, can be used to turn unification into matching, as I mentioned briefly in Chapter 4. There, I pointed out that this was desirable to reduce the number of solutions suggested by unification.

There is a further reason why the unification operation must be reduced to its constituent parts of unifying corresponding subterms. This relates to the use of embedded control information when higher-order unification will be applied. Some of the functions the embedded notation performs are effected by first-order unification in our usual first-order systems. These functions are not so straightforwardly realised by the more complex higher-order algorithm, as I shall explain.

The usage we make of wavefronts to record the status of the conclusion through its components is essentially the same as in regular CLAM. Our use to choose suitable wave rules is also the same except that the unification process is changed to identify meta-variables, and some complications arise here.

This second process encompasses several activities, which are performed simultaneously by first-order unification, but must be distinguished for higher-order unification. This is necessary firstly because the higher-order algorithms cannot handle the notation being present in the terms, due to typing prob-

lems, as already noted in the previous chapter. Secondly, even if the notation could be left *in situ*, it would not perform the functions we wish of it when a higher-order unification algorithm was applied. Therefore functions which the wavefront notation performs implicitly in first-order unification must now be performed explicitly.

The functions of wavefronts in choosing wave rules to match an expression are:

- to align the use of the rule, so that its components are fitted to the corresponding components in the expression, so wavefronts match wavefronts etc.

- to identify types of rules (transverse or longitudinal) and directions of wavefronts. This identification may be used either to select appropriate rules or to inform the planner as to what has been found.

In first-order unification, wavefronts can only match each other. We can just leave the wavefronts in place and unification does all this. Higher-order unification is less helpful. It would instantiate variables to wavefront notation, or use projection to ignore the notation. We have to do our own alignment, identification and selection.

My solution to all these problems is to unify progressively from the holes to larger and larger terms.

6.3.3 Inadmissible Unifications

In Chapter 4, it was noted that unification may suggest substitutions for meta-variables, but some of these unifications, although perfectly satisfactory for unification purposes, will not be sensible substitutions. These substitutions are barred by temporal scoping. Here is an example of how this arises.

Suppose, for example, we have the following meta-sequent:

1. $h \in pnat$

2. $t \in pnat\ list$

3. $\forall a \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} F(reverse(t), a)$

$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} F(\boxed{append(\underline{reverse(t)}, h :: nil)}^{\uparrow}, a)$

where F is a meta-variable to be instantiated through the demands of the subsequent proof. The wave front information shows how rippling is progressing, and tells us how to align this term with a wave rule to effect suitable rewritings. For reasons that will be explained in chapter 7, the next proof step is to use a special version of the associativity of *append* lemma, and we expect to identify F by considering unifications of

$$append(append(reverse(t), h :: nil), a)$$

with

$$F(append(reverse(t), h :: nil), a)$$

Higher-order unification suggests four unifiers for F:

i. $\lambda u \lambda v.append(u, v)$

ii. $\lambda u \lambda v.append(u, a)$

iii. $\lambda u \lambda v.append(append(reverse(t), h :: nil), v)$

iv. $\lambda u \lambda v.append(append(reverse(t), h :: nil), a)$

At this point in the proof, each of them would be a valid object to use for F *in terms of Oyster's logic*, as well as in terms of the unifying algorithm. The last three are not sensible choices, because they build current universally quantified or free variables (a, t) into the identity of the function, F. As F must have been identifiable without reference to these internal names, prior to proof, they break the rules of temporal scoping. For that reason, my system discards them. In doing this it is using meta-knowledge about the type of problem being solved.

Another restriction useful in pruning unifiers would be a notion of generality. Some unifiers specify constant terms directly in order to unify, rather than giving a more general unifier using arguments. E.g. $F(0)$ can be unified with $f(0)$ either by letting F be $\lambda x.f(0)$ or $\lambda x.f(x)$. Although the former must always be acceptable, it is unlikely to be preferable, as we would normally choose the more general solution.

6.4 Normalisation

The expressions resulting from MOR may require some λ−conversion to ensure that subsequent methods are not impeded from applying. Any such normalisation of a conclusion and its corresponding induction hypothesis must be co-ordinated.

An example of this need for normalisation occurs in the base case of the proof from which the sequent above was extracted. F becomes instantiated to $\lambda u \lambda v.append(u, v)$. The base case of the proof is then:

$$\vdash \forall a \in pnat\ list\ \exists y \in pnat\ list.\ y =_{pnat\ list} \lambda u \lambda v.append(u, v)\ of\ reverse(nil)\ of\ a$$

Oyster's base method applies to $reverse(nil)$, enabling it to be rewritten to nil. It does not apply to the conclusion resulting from this, involving a non-β-reduced term. The solution I adopted was to β-reduce any instantiated expression before trying to use a stored lemma to rewrite it. Use of stored lemmas on expressions not involving meta-variables only uses first-order matching, so this normalisation would not routinely be done by standard CLAM.

6.5 Implementing MOR with Methods

I implemented MOR within the CLAM system. This allowed me to experiment with various interesting "how" problems. The approach I took was to build some new methods to introduce the generalisations appropriate to the particular problems, using meta-variables as required for MOR. Some existing methods underwent major changes to take account of the presence of meta-variables. I shall describe these in the subsequent chapters devoted to the different generalisation problems undertaken. Other existing methods were adapted in minor ways to take account of the possible presence of meta-variables. I will describe them below, since the purpose of these changes was largely negative, to stop methods applying inadvertently, when insufficiently instantiated.

I used CLAM's own search control of applying a list of methods preferentially depth-first or via iterative deepening to control reasoning using meta-variables. This was effective in instantiating meta-variables, and is described in detail, in the chapters on the generalisations tackled.

Representing meta-variables by Prolog variables has the advantage that no extraneous term structure is inserted into sequents to label these variables explicitly, and using Prolog to instantiate them propagates values appropriately wherever those Prolog variables are in the meta-proof. The disadvantage is that they are *not* explicitly labelled, and therefore become invisible with respect to MOR, once they have been instantiated.

6.5.1 New Methods

I designed new methods to introduce particular generalisations involving meta-variables for MOR. They were incorporated into the same framework as the rest of CLAM's methods. These are described in detail in the subsequent chapters.

6.5.2 Inhibiting Methods with Inadequately Specified Input

Some restrictions are necessary to control mis-application of methods when meta-variables are present. This is achieved by augmenting the preconditions to require the absence of meta-variables. Such restrictions are implemented in the following methods:

Symbolic Evaluation

The base and reduction submethods appear as part of CLAM's symbolic evaluation method, early in its preference list of methods. The adaptation which instantiates functional meta-variables is mainly in the wave method, and also the step submethod, later in the list. It would be sufficient to make this restriction only apply to higher-order meta variables, but it never encounters any others.

Strong Fertilization

Strong fertilization is a highly preferred method to apply, since it reduces the complexity of the conclusion significantly. It appears early in CLAM's preference list, well before the wave method and step submethod which can instantiate functional meta-variables. It must not be applied before these variables have been instantiated, because it could cause the meta-variables to vanish before instantiation had occurred. Returning to the sequent above:

1. $h \in pnat$
2. $t \in pnat\ list$
3. $\forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} F(reverse(t), a)$

   $\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} F(append(reverse(t), h :: nil), a)$

Strong fertilization would apply easily, since the a's and y's could be introduced corresponding to each other, and the F terms are unifiable, as a flexible-flexible pair.

The result would be quite useless. The induction would be invalid, as the measure which ensures induction is well-ordered would not have decreased. The proof plan branch would terminate without ever having identified F.

Existential

This, too, was restricted so as not to apply when meta-variables were already present. Controlling two speculations at once is hard.

6.5.3 Changing the Planning Search Space

A simple piece of search control is the alteration of the order of generation of subgoals in the *planning space*, so that the planner finds the informative ones first. The order of goals in the proof space need not be affected. This is a crude static version of the kind of dynamic control of search for plans described below. It is used in the induction method, where CLAM normally generates base and step subgoals for the planning space in that order, the same order that Oyster uses. This is unfortunate for the planner, because the lemmas which apply rewrites in base cases match more easily and less informatively than those in step cases. Once $reverse(nil)$ has been rewritten to nil, the base case of the example I have used above is

$$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} F(nil, a)$$

Many definitions of functions over two lists would match F, mostly wrongly, and cause unnecessary backtracking. My adapted induction method generates the step subgoal(s) first, so that the planner encounters them first.

6.6 Middle-Out Planning

A desirable extension to the whole system would be middle-out planning. There are two main abilities such a planner should have:

- The creation of plans with variable components - i.e. with gaps left in for inductions of unknown structure, or for case splits or some other inference which might be needed later.

- Use of different search strategies. Currently, the planner can operate depth-first, breadth-first, via iterative deepening or as guided by a limited heuristic mechanism. It would be useful to be able to direct the planning to extend the meta-proof as instantiation takes place, postponing nodes which aren't providing such information, and come back to them when progress has been made and the information has, at least partially, instantiated the uninformative node. The search would be driven dynamically by successful instantiation of meta-variables, rather than by a fixed strategy chosen in advance.

  A simple example of this is the step case/base case swop described in the previous section. Plenty of base case rules could apply, but the information present to select between them is slight.

6.7 Conclusion

MOR is a useful extension to the existing planning system. With care and use of proof structural information, all the existing functionality can be preserved. We are able to attempt new problems, but different kinds of problem make differing demands on the planning as well as the reasoning parts of CLAM. Reasoning may be performed at the meta-level, which instantiates unknown quantities in

a proof "on demand", and the instantiated plan can then create a valid object level proof, and hence an extract term.

Chapter 7

Synthesising Tail-Recursive Functions from Naïvely Defined Specifications

In this chapter I will describe how *MOR* can be used to synthesise tail-recursive functions from given naïvely recursive specifications. Most attempts at generating tail-recursive programs transform an existing program, perhaps guided by templates. Characterising tail-recursion through proof structures provides a means of guiding synthesis of the corresponding programs.

This is an interesting problem which can be captured by generalisation. It involves finding a way of accumulating the work done by the naïve function at each stage rather than stacking it up to be executed.

To try to make this palatable, I will describe the work in several stages, and illustrate it with examples. In section 7.1 I will compare naïve and tail-recursive versions of execution for list reversal and relate this to the structure of the corresponding proof and extract term. Then, in section 7.2 I will discuss the characterisation of the synthesis of tail-recursive functions through proof. In the light of this characterisation, I will show how a tail-recursive algorithm may be found from a naïve one (7.3), and how these proofs are structured (7.4). In section 7.5, I will explain the adaptations to CLAM to let it use *MOR* at the various stages where it is needed to find suitable proofs and synthesise. Then section 7.6 will show how the search for a proof is controlled with reference to

the (by now familiar) *reverse* example. Lastly I will give a list of functions this system can synthesise, and draw conclusions about the merits of the approach.

When I carried out this work on tail-recursion, I used the version of CLAM current at the time, not the newest version, which has only become available since. In my description, I will revert to the notation available in the version I used, i.e. omitting directions of wavefronts and annotations marking sinks.

7.1 Comparison of Naïve and Tail-Recursive Reverse

7.1.1 Naïve Reverse

Suppose we have a definition[1] using naïve reverse, so that we know that:

$$\forall x \exists y.\ y =_{pnat\ list} reverse(x) \tag{7.1}$$

where $reverse(x)$ is defined as:

$$list\_ind(x, nil, [h, t, r, append(r, h :: nil)])$$

As a reminder of what this looks like in more familiar terms:

$$\begin{aligned} reverse(nil) &== nil \\ reverse(h :: t) &== append(reverse(t), h :: nil) \end{aligned}$$

This corresponds to a function which when given the list $h_1 :: h_2 :: \ldots$ to reverse, will compute *reverse*'s definition to construct the term

$$append(reverse(h_2 :: \ldots), h_1 :: nil)$$

---

[1]This formulation is appropriate in the constructive type theory framework because it yields an executable extract term corresponding to the witness for the existence of y, i.e. the naïve function.

and then again to construct the term

$$append(append(reverse(\ldots), h_2 :: nil), h_1 :: nil)$$

and so on:

$$append(append(append(..., ...), h_2 :: nil), h_1 :: nil)$$

which it will be unable to evaluate until the end of the t list, and $reverse(nil)$ is reached. This is what is implied by this version being naïvely recursive. Each call to the function cannot complete its evaluation because it is waiting for one or more of its subordinate calls to complete. A stack of these builds up.

This stacking mirrors an underlying machine implementation. I will give a simplified account assuming a call-by-need or lazy evaluation, where only those arguments are evaluated which are necessary for the outermost function to be evaluated. Suppose a function call is in a frame at the top of the stack. The operation of the machine is to take such calls, evaluate them, and replace them by the result of their evaluation. $reverse(h_1 :: h_2 :: \ldots)$ is defined in terms of *append*:

$append(reverse(h_2 :: \ldots), h_1 :: nil)$
STACK

but this call cannot be evaluated without evaluating the *reverse* inside it, and another call is placed on the stack above the first one:

$append(reverse(\ldots), h_2 :: nil)$
$append(reverse(h_2 :: \ldots), h_1 :: nil)$
STACK

... and so on, since this contains yet another call which must be evaluated. The stack is forced to grow, clogged up with function calls waiting to complete their execution. Time is spent on the management of a larger stack.

7.1.2 Tail-Recursive Reverse

A tail-recursive version, *reverse2*, would constantly accumulate the reversed portion of the list traversed so far. Using an accumulator, a, this is defined as:

$$\begin{aligned} reverse2(nil, a) &== a \\ reverse2(h :: t, a) &== reverse2(t, h :: a) \end{aligned}$$

When a is taken to be nil, *reverse2* reverses a list in the first argument position. To compute $reverse2((h_1 :: h_2 :: \ldots), nil)$, the equivalent steps to the naïve computation above are:

$$reverse2((h_2 :: \ldots), h_1 :: nil)$$

then

$$reverse2((\ldots), h_2 :: h_1 :: nil)$$

and so on. There is no need to use more stack frames, as each call can be completed and replaced by its recursive call to itself.

Diagrammatically, the machine initially has the following stack:

$reverse2((h_2 :: h_1 :: \ldots), nil)$
STACK

and it can *replace* the top frame as follows:

$reverse2((h_1 :: \ldots), h_2 :: nil)$
STACK

and then:

$reverse2((\ldots), h_1 :: h_2 :: nil)$
STACK

Although the same number of recursive calls are used, the space required by the tail-recursive algorithm is much smaller. The advantage is not only that such an algorithm takes up less stack space, but that it can be transformed into a program without any recursion at all, i.e. one which is iterative.

The definition of tail-recursion in computing terms is that the recursive definition of the function calls on itself only once, and that that call is the last

action in the body of the definition. This does not mean that the definition of the function may only refer to itself once. Multiple references are allowed if each is within a separate branch of a conditional, and is the last action in its branch.

7.1.3 A "Tail-Recursive" Theorem

The theorem generalising 7.1 which can yield a tail-recursive function is

$$\forall x \forall a \exists y.\ y =_{pnat\ list} append(reverse(x), a) \tag{7.2}$$

append works to accumulate the list processed so far onto the accumulator a, achieving much the same effect as *reverse2*, above.

Thus formulated, the construction of the witness for y can provide the (tail-recursive) function we are interested in. It must satisfy the definition of tail-recursion given above. By speculating the form a suitable generalisation would take, and circumscribing its subsequent proof so that the construction of y's witness meets the definition of tail-recursion, we can synthesise a tail-recursive function for a specification.

The difficulty for an automatic system, given 7.1 is in guessing what the generalisation is, and what the accumulating function (*append* in 7.1.3), should be. The structure of the proof provides information, which can be used by MOR to generate the answer to this question. In the next sections I will explain how this works and what the search control problems are.

7.2 Characterising The Synthesis of Tail-Recursive Functions

As I have just described, tail-recursive functions are those which are defined so that an evaluation of a call to the function may be completely replaced by another recursive call on itself, with different arguments. This is apparent in the reverse example above. In the naïve version, the evaluation is continually

postponed until all the list has been unravelled. In the tail-recursive version, each evaluation is replaced by a recursive call where the recursion argument is reduced. There are still as many recursive calls in either version, but the tail-recursive one can be implemented as a *while* loop.

In the constructive logic framework, tail-recursion is determined by the way we use the induction hypothesis. The witness for this hypothesis is the construction corresponding to the function's recursive call on itself. If we are able to use it (or one of its subterms) alone as the justification of the goal, then no further work is required after the recursive call, and the function is tail-recursive. If instead we have to embed it in some function, that function application enters into the construction, which is no longer tail-recursive.

Although in my definition of tail-recursion, I mentioned the possibility of conditionals, that is a complication I do not address in full in these proof structures. The functions synthesised here involve primitive conditionals inherent in recursive definitions, such as whether a number is 0 or $s(\_)$, and whether a list is *nil* or non-empty. Splitting into such conditional branches happens naturally as a result of induction. More complex conditions, not arising as part of the induction, and necessitating a case split, fit the proof requirements described, but are beyond the capabilities of the planning implementation at present. As I explained in chapter 6, introducing case-splits requires more advanced planning. Such extended $\mathcal{MOR}$ and middle-out planning involving conditionals is an avenue for further work, see chapter 11.

I shall illustrate the construction of the recursive step by contrasting the crucial steps in the two proofs: of the theorem which corresponds to a naïvely recursive algorithm, and the one that corresponds to a tail-recursive algorithm.

7.2.1 Naïve Reverse

This proof is artificial, because it synthesises a function we already know, but it is informative for comparison with the tail-recursive version.

The theorem is

$$\vdash \forall x \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$$

Introduction of x followed by induction gives the following step case:

1. $x \in pnat\ list$
2. $h \in pnat$
3. $t \in pnat\ list$
4. $ih_n \in \exists y.y =_{pnat\ list} reverse(t)$

$\vdash \exists y.y =_{pnat\ list} reverse(\boxed{h :: t})$

The extract term corresponding to the whole proof is $\lambda x.list\_ind(x, \_, [h, t, ih_n, \_])$, where the gaps are to be filled by the extract terms from the base and step proofs respectively. Elimination on the induction hypothesis grows the proof tree to this node:

5. $y' \in pnat\ list$
6. $ih'_n \in y' =_{pnat\ list} reverse(t)$
7. $ih^e_n \in ih_n =_{\exists y.y=_{pnat\ list} reverse(t)} y'\&ih'_n$

$\vdash \exists y.y =_{pnat\ list} reverse(\boxed{h :: t})$

Note that using the induction hypothesis has given us those entities that it witnesses - it breaks down into a pair $y'\&ih'_n$ (7), which is a list y' (5), and y' is the reverse of t, as proved by ih'_n (6). I will sometimes elide hypotheses such as (7) in sequents, when they do not actively contribute to the proof, and they obscure the presentation. The extract term grows accordingly:

$$\lambda x.list\_ind(x, \_, [h, t, ih_n, spread(ih_n, [y', ih'_n, \_])$$

Longitudinally rippling the conclusion using the definition of reverse[2] changes the conclusion, but not the extract term:

---

[2]Unfolding and folding according to definitions have no effect on the extract term because they only affect the representation preferred by the human user.

$$\vdash \exists y.y =_{pnat\ list} \boxed{append(\underline{reverse(t)}, h :: nil)}$$

Now this key branch of the proof can be completed. The object introduced for y is $append(y', h :: nil)$, and extract term is further instantiated to

$$\lambda x.list\_ind(x, \_, [h, t, ih_n, spread(ih_n, [y', ih'_n, append(y', h :: nil)\&\_])$$

The two gaps are, respectively, for the base case and the proof that the introduced object is suitable, i.e. that

$$append(y', h :: nil) =_{pnat\ list} append(reverse(t), h :: nil)$$

In the step case, the essential part of the recursion has now been built. Significantly, it wraps an extra function call ($append(...,...)$) which will have to be executed at each recursion, around the recursive one (y').

7.2.2 Tail-Recursive Reverse

I shall now describe a proof for the corresponding tail-recursive version of reverse and compare them. The proof starts off similarly to the one above, x is introduced, used for induction, and the definition of *reverse* is used to ripple the conclusion:

1. $x \in pnat\ list$
2. $h \in pnat$
3. $t \in pnat\ list$
4. $ih_t \in \forall a' \exists y.y =_{pnat\ list} append(reverse(t), a')$

$$\vdash \forall a \exists y.y =_{pnat\ list} append(\boxed{append(\underline{reverse(t)}, h :: nil)}, a)$$

The extract term corresponding to the whole proof is $\lambda x.list\_ind(x, \_, [h, t, ih_t, \_])$ as before. Although neither Oyster nor CLAM would distinguish the names of the accumulator, a, in the induction hypothesis and conclusion, the fact that they are distinct is crucial to this solution. To emphasise that, I have renamed the hypothesis a as a'.

Unlike in the previous proof, it is now possible to use associativity of *append* to ripple the wavefront transversely. This is a preparation for the following stage.

1. $x \in pnat\ list$
2. $h \in pnat$
3. $t \in pnat\ list$
4. $ih_t \in \forall a' \exists y.y =_{pnat\ list} append(reverse(t), a')$

$\vdash \forall a \exists y.y =_{pnat\ list} append(reverse(t), \boxed{append(h :: nil, \underline{a})})$

This substitution justified by the associativity of *append* has no effect on the extract term. As for any substitution, it results in a proof branch to justify that the terms involved in the substitution were equal.

The definition of append simplifies the conclusion, again without affecting the extract term:

$\vdash \forall a \exists y.y =_{pnat\ list} append(reverse(t), \boxed{h :: \underline{a}})$

Now we take advantage of the fact that the two a's acting as accumulators may be different. We can instantiate a' to be whatever term causes the induction hypothesis to match the conclusion.

Introducing a free a for the universal variable in the conclusion, and instantiating a' to $h :: a$, the sequent becomes:

$a \in pnat\ list$

$ih_t^a \in \exists y.y =_{pnat\ list} append(reverse(t), h :: a)$

$\vdash \exists y.y =_{pnat\ list} append(reverse(t), h :: a)$

For the extract term, these steps correspond to introducing a λ-term to the extract term, and, since universal quantification is identified with the dependent function, to a substitution to be applied to the extract term below this node (denoted by the subscript in curly brackets). The extract term has become:

$$\lambda x.list\_ind(x, \_, [h, t, ih_t, \lambda a.\__{\{ih_t(h::a)/ih_t^a\}}])$$

The crucial difference which makes this proof synthesise a tail-recursive function happens next. The witness arising from the induction hypothesis can be used *directly* for the conclusion, without involving any other function applications. This is the essential feature which guarantees that the function extracted will be tail-recursive with respect to this step.

ih_t^a can now be introduced directly as evidence for the conclusion, completing the step case definition. Unlike the naïvely recursive version, it is not introduced as part of a term:

$$\lambda x.list\_ind(x, \_, [h, t, ih_t, \lambda a.(ih_t^a)_{\{ih_t(h::a)/ih_t^a\}}])$$

The substitution is applied automatically, making this:

$$\lambda x.list\_ind(x, \_, [h, t, ih_t, \lambda a.ih_t(h :: a)])$$

The distinguishing feature of a tail-recursive synthesis has been achieved. A hypothesis which is an instance of the induction hypothesis has been used alone to provide the witness for the existential conclusion.

7.3 Building a Tail-Recursive Algorithm from a Naïve One

The example in the previous section illustrates the difference between these types of algorithm. By considering this for an arbitrary function f, we can identify a general pattern which can be exploited to synthesise tail-recursive algorithms from naïve ones.

7.3.1 Naïve Operation of Primitively Defined Functions

We will suppose that f is primitive recursive on its first argument,

$$f(\boxed{c(\underline{x})}, \vec{z}) == \boxed{f'(\underline{f(x, \vec{z})}, x, \vec{z})}$$

where c represents some recursive constructor, such as successor, or 'cons'ing a head onto a list. I use $\vec{z}$ to denote arbitrary other arguments, although there need not be any.

Notice that this pattern corresponds to our notion of a longitudinal wave rule, where some portion of a term is moved upwards through an unchanged surrounding term. Similarly to the *reverse* case, attempting to evaluate f results in a stack of calls to f' building up.

7.3.2 Tail-Recursive Operation

The tail-recursive version described here also unfolds its definition upon its arguments, but uses an accumulator to store the value of the computation so far. This value is carried forward into the recursive call, which *is the result* of the original call.

One must be cautious in saying this, though, as depending on how much work is done to simplify the accumulating expression, what is accumulated may still require evaluation itself. For example one might build up and carry around an accumulated term like

$$\ldots append(h_n :: nil, append(h_{n-1} :: nil, \ldots, nil)) \ldots$$

and evaluate it at the end instead of at each stage. This may be less efficient but it would still be tail recursive. It could still have reduced demands on the stack, since such argument values would probably be stored via a reference to some memory location.

With or without a simplified accumulator, tail-recursive evaluation is not postponing a stack of function calls which take up space. At each recursion, the current recursive call applies to the recursion argument and accumulates onto the processing that is completed. When the recursion ends, the accumulator holds the result.

Some functions, such as *reverse2*:

$$reverse2(\boxed{h :: \underline{t}}, a) == reverse2(t, \boxed{h :: \underline{a}})$$

are defined so as to build this action in immediately.

This is the kind of function we want to synthesise, but instead of defining a new tail-recursive function, such as *reverse2*, with an extra accumulator argument, we build its equivalent compositely. We achieve this by:

- taking the original function definition as a specification, and
- using this specification in a proof structure which enforces tail-recursiveness. I.e. one which takes the wavefront and accumulates it. We add an accumulator and functions to do the accumulation. I.e. some a and g, such that

$$g(\boxed{f'(\underline{f(x,\vec{z})},x,\vec{z})},a) == g(f(x,\vec{z}),\boxed{f''(\underline{a},x,\vec{z})})$$

This characterisation describes a tail-recursive function, while tying it to the original function's definition. Not all formulations of tail-recursion require accumulators, I will return to this point later.

As I indicated at the start, the problem lies in guessing what the accumulating functions, g and f'', are. Although no general way is known of generating them automatically, we can describe the structure of the associated synthesis proofs. Constructing such a proof may indicate what these unknown functions must be.

7.3.3 Making the Specification Guide the Proof

Let us take an initial goal stating that a value y can be constructed which is equal to the value produced by the naïve specification, f:

$$\forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \exists y \in t.\ y =_t f(x,\vec{z}) \tag{7.3}$$

Provided we can come up with a function which computes y, and satisfies the equality, we know that it, too, meets the specification. If we follow our noses, as in 7.2.1, we will just get a naïve function, like the specification. If we generalise the theorem to build in an accumulator and suitable functions, we can turn out a tail-recursive function, as I have just described. I say "generalise", firstly

because the function encapsulated here is more general than the original, just as *reverse2* is more general than *reverse* - it covers a wider range of values, but on a subset of them, can cover all the functionality of the original. Secondly, because we will expect this new goal to be a hypothesis used to imply the original, so in logical terms it is a generalisation.

At the meta-level, the new goal proposed is:

$$\forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.\ y =_t G(f(x, \vec{z}), a) \tag{7.4}$$

$\mathcal{MOR}$ can be used to guide the choice of the function G.

The types of G and a can easily be inferred. a is to accumulate an object which can be the result of an evaluation of f. Since f produces objects of type t, a must have type t. G is then obliged to be a function which takes two terms of type t, and produces another one, since it is an argument of $=_t$. The new universal quantifier for a should follow all the other ones, since when it is used, it may be for a term involving them.

By using the cut rule, we split the proof into two branches:

- To prove the generalised goal corresponding to the tail-recursive algorithm. The planning should identify a G which gives us a tail-recursive algorithm. The extract term resulting from this will execute tail-recursively.

- To prove that the original goal can be justified by the generalised goal, guaranteeing that the tail-recursive function gives the same results as the naïve one. This must include specialising the more general function for the set of values where it includes the original function. I.e. in the *reverse2* case, where the accumulator starts at *nil*.

In the next section, I will explore how the proof can identify G. I will describe this first for the *reverse* example and then more generally. Subsequently, I will describe how such proofs are selected by planning from the initial naïvely defined goals.

7.4 The Structure of Tail-Recursive Synthesis Proofs

7.4.1 Tail-Recursive Reverse

The proof proceeds largely as in 7.2.2. The generalised goal is:

$$\vdash \forall x \in \mathit{pnat\ list} \forall a \in \mathit{pnat\ list} \exists y \in \mathit{pnat\ list}.\ y =_{\mathit{pnat\ list}} G(\mathit{reverse}(x), a)$$

Induction

Recursion analysis suggests x as a candidate for induction, since it is a universally quantified variable occurring in the recursive argument position of *reverse*, and so a suitable wave rule exists for subsequent rewriting. The definition of *reverse* suggests simple list induction.

The result of the induction is two new goals, the base:

$$\vdash \forall a \in \mathit{pnat\ list} \exists y \in \mathit{pnat\ list}.\ y =_{\mathit{pnat\ list}} G(\mathit{reverse}(\mathit{nil}), a)$$

and step cases of the induction:

1. $h \in \mathit{pnat}$
2. $t \in \mathit{pnat\ list}$
3. $\forall a \in \mathit{pnat\ list} \exists y \in \mathit{pnat\ list}.\ y =_{\mathit{pnat\ list}} G(\mathit{reverse}(t), a)$

   $\vdash \forall a \in \mathit{pnat\ list} \exists y \in \mathit{pnat\ list}.\ y =_{\mathit{pnat\ list}} G(\mathit{reverse}(\boxed{h :: \underline{t}}), a)$

The base case is too uninstantiated to be informative, unlike the step case.

Longitudinal Rippling

It is now possible to use the definition of *reverse* to move the wavefront upwards in the term structure:

$$\vdash \forall a \in \mathit{pnat\ list} \exists y \in \mathit{pnat\ list}.y =_{\mathit{pnat\ list}} G(\boxed{\mathit{append}(\underline{\mathit{reverse}(t)}, h :: \mathit{nil})}, a)$$

Transverse Rippling

So far, all the steps have been unaffected by the presence of the meta-variable, G. Here, though, comparing with the earlier proof, we would expect to use the associativity of *append* to move the wavefront sideways so that the conclusion can be justified by the induction hypothesis. If G is identified as $\lambda u \lambda v.append(u, v)$, associativity can be used in exactly the same way, and the meta-level proof continues as before, with G instantiated. Notice that the $\lambda-$variable arguments are in an order determined by the unification to match the function in the wave rule. This is achieved by extending the wave method to use higher-order unification, as described in chapter 4, and section 7.5.

Fertilization

1. $h \in pnat$
2. $t \in pnat\ list$
3. $\forall a \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} append(reverse(t), a)$

   $\vdash \forall a \in pnat\ list$

   $\exists y \in pnat\ list.\ y =_{pnat\ list} append(reverse(t), \boxed{append(h :: nil, \underline{a})})$

Fertilization now applies.

This a little different from the tidier proof I showed earlier, as it fails to simplify the argument in the accumulator position. Ideally, we would have simplified the $\boxed{append(h :: nil, \underline{a})}$ subterm first. Without wishing to get too side-tracked into a discussion of planner control, I should point out some problems with achieving such a simplification. Firstly, CLAM applies any method which can terminate a branch, preferentially, so fertilization will apply before the symbolic evaluation which would perform this simplification. Secondly, this is not a simple reduction, it needs two uses of the different parts of the definition of append. Thirdly, it is really a special case of having the power to *ripple in* - rippling wavefronts down into an accumulator, which has only been implemented in recent,

experimental versions of CLAM. Without directional information this is hard to control. Fourthly, it is possible to force rippling in the old version by abandoning loose coupling of CLAM methods under the planner's control in favour of a strictly defined supermethod, like the overall induction strategy.

Base Case

Once G has been identified as *append*, the base case is handled by the symbolic evaluation method:

$$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} a$$

the existential method:

$$\vdash Y =_{pnat\ list} a$$

and finally the "tautology" method, which handles simple equalities.

Justification

1. $j \in \forall x \in pnat\ list$

$$\forall a \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} append(reverse(x), a)$$

$$\vdash \forall x \in pnat\ list \exists y \in pnat\ list.\ y =_{pnat\ list} reverse(x)$$

The universally quantified x from the conclusion is introduced and its equivalent in the hypothesis is eliminated, instantiating it to the same thing:

2. $x \in pnat\ list$
3. $j' \in \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(x), a)$

$$\vdash \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$$

A value, a_0, for a must now be produced. Since we are expecting to use the y generated by the hypothesis for the conclusion, a_0 must be such that:

$$reverse(x) =_{pnat\ list} append(reverse(x), a_0) \tag{7.5}$$

This equality guarantees that the synthesised function is everywhere equal to the original.

By eliminating j' taking nil for a_0, and eliminating the resulting hypothesis again to produce a value y' for y, we reach this sequent:

4. $j'_{nil} \in \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(x), nil)$
5. $y' \in pnat\ list$
6. $j''_{nil} \in y' =_{pnat\ list} append(reverse(x), nil)$

$\vdash \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$

We select y' to introduce for y, and then the symbolic evaluation and the "tautology" method complete this branch of the planning proof.

This branch is not straightforward without meta-knowledge about the generalisation hypothesis. We need to know which hypothesis is the generalisation, and what value to initialise the accumulator to.

7.4.2 General Tail-Recursive Synthesis

Having examined the automation of the synthesis proof for the particular case of the *reverse* function, I shall now look at it in schematically, and discuss some of the problems of planning the various stages. There is no guarantee that all these steps will work in any particular case, but we can look at the shape of what is happening.

Starting with the original goal as:

$$\vdash \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \exists y \in t.y =_t f(x, \vec{z})$$

the generalised goal is:

$$\vdash \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t G(f(x, \vec{z}), a)$$

Induction

Recursion analysis suggests x as a candidate for induction: The definition of f suggests induction based on x, with step case defined on $c(x)$.

The result of the induction is some base and step goals. The latter are more informative, suppose one of them is:

1. $x \in t_x$
2. $\forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t G(f(x, \vec{z}), a)$

   $\vdash \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t G(f(\boxed{c(\underline{x})}, \vec{z}), a)$

Longitudinal Rippling

Using a longitudinal wave rule:

$$f(\boxed{c(\underline{X})}, \vec{Z}) \rightarrow \boxed{f'(\underline{f(X, \vec{Z})}, X, \vec{Z})}$$

A longitudinal ripple takes place:

$$\vdash \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t G(\boxed{f'(\underline{f(x, \vec{z})}, x, \vec{z})}, a)$$

Potentially, there could be further longitudinal ripples here and G would become partially instantiated. This would be difficult for us to handle at the moment, because it would require us to think of G as a composite function, $G'(G''(\ldots))$, and only to instantiate G''. In the absence of knowing the types of G' and G'', the unification algorithm is unable to do this.

However, it is hard to see many cases when this might be needed, as any such G'' would almost certainly have to be monadic. Access to the accumulator would be only via the topmost part of the function, or there would be little point in having it.

Transverse Rippling

Using a transverse ripple of the form:

$$g(\boxed{f'(\underline{f(X,\vec{Z})},X,\vec{Z})},A) \rightarrow g(f(X,\vec{Z}),\boxed{f''(\underline{A},X,\vec{Z})})$$

we move the wavefront sideways onto the accumulator so that the conclusion can be justified by the induction hypothesis. G is instantiated to g, throughout the meta-level sequent:

1. $x \in t_x$
2. $\forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t g(f(x,\vec{z}),a)$

   $\vdash \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t g(f(x,\vec{z}),\boxed{f''(\underline{a},x,\vec{z})})$

This completes the instantiation of G.

Although this step is what we're aiming for, we should not make it higher priority than longitudinal rippling. Since the new accumulator was wrapped around the outside of the original term, we expect to ripple longitudinally to reach it. Allowing transverse ripples earlier would permit other universally quantified variables to function as accumulators and be rippled into. If we could do that anyway, we should not have attempted to add another accumulator.

A further possibility which has only become available in the new versions of CLAM is wave annotation which records directions of wavefronts. This would now make it possible to direct wavefront movement longitudinally up the term tree from the induction variable and then longitudinally down into the accumulator, Currently the system can only move the wavefront up and then transversely onto the accumulator. A supermethod could keep track of wavefronts explicitly and control the direction of wavefronts like this itself. This has not proved necessary for any of the problems I have attempted.

Fertilization

The derived induction hypothesis now implies the current conclusion, and fertilization can take place.

Notice that the order of the quantifiers is important. The accumulator should be last, to be able to instantiate to anything involving any of the other universally quantified variables.

Base Case

As in the *reverse* example, once G has been identified, routine steps of the symbolic evaluation, existential and tautology methods handle this.

Justification

1. $j \in \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.\ y =_t g(f(x, \vec{z}), a)$

   $\vdash \forall x \in t_x \forall \vec{z} in t_{\vec{z}} \exists y \in t.\ y =_t f(x, \vec{z})$

Each universally quantified variable in the conclusion is introduced. For each of these, the corresponding one in the justifying hypothesis is instantiated to follow it, until there are no more in the conclusion:

n. $j_n \in \forall a \in t \exists y \in t.\ y =_t g(f(x, \vec{z}), a)$

   $\vdash \exists y \in t.\ y =_t f(x, \vec{z})$

As in the *reverse* example, a suitable value is picked for a, and the branch completed using the symbolic evaluation, existential and tautology methods.

To preserve the tail-recursion which has been constructed, it is important that the same witness is used for the existential variable in the conclusion as in the hypothesis. Otherwise, if a surrounding function were built on, tail-recursiveness would be lost.

7.4.3 Picking a Base Value for the Accumulator

We need an a such that

$$f(x) \quad =_t \quad g(f(x), a) \tag{7.6}$$

Finding a suitable value for a such that (7.6) is true could require arbitrary amounts of theorem proving, but a feasible means of doing so for most cases is to use further MOR to look for a lemma in the CLAM library of the form

$$\forall x.\ g(x, A) = x$$

and take the value A becomes instantiated to for a. Since such a lemma is likely to be stored as a useful reduction rule, we have a good chance of finding a value this way.

7.4.4 Overall Pattern

The overall pattern is:

tail-recursive generalise

induction	justification
symbolic evaluation	wave (one or more longitudinal)
existential	wave (transverse)
tautology	fertilize

7.5 Adaptations to CLAM for MOR

Having seen what must be achieved by the planner, I'll turn to examining how it does it.

As described in chapter 5, CLAM uses methods to describe the preconditions and effects of proof components. I took the approach of modifying some of these and adding new ones for particular experiments, taking advantage of the existing framework.

Most of the methods are largely unchanged, except to inhibit their operation when insufficiently instantiated, as described in chapter 6, or to add beta-reduction, to normalise functions which have been introduced as $\lambda-$functions. A new tail-recursive generalisation method was introduced, major changes took place in the wave method and minor changes were made to restrict the existential method. I will describe these three in detail and their operation in conjunction with the planning mechanism.

Allowing the methods to operate independently like this works for straightforward examples. It seemed an attractive option to start with, as it followed one of the ideas of CLAM, that the methods have an independent validity, and should be able to operate linked by the planner under the control of their preconditions.

In this case though, using independent methods sometimes throws too much knowledge about the proof branches away. Eventually I built a compound supermethod, embodying a strategy incorporating several methods, to perform tail-recursive generalisations. This also became necessary because the inhibitions built into the adapted existential method, to stop it producing trivial unwanted solutions, became harder and harder to define so that they would impede the method only in the right places, without meta-knowledge of the current proof stage. This will also be described, after the free-standing version.

7.5.1 Tail-Recursive Generalisation

This new method detects a specification sequent, and uses the cut rule to introduce a generalisation which $\mathcal{MOR}$ will use to synthesise a tail-recursive function.

Preconditions

The method applies when the goal is of the form

$$\forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \exists y \in t_y.\ y =_{t_y} f(x, \vec{z})$$

The $\vec{z}$ may be absent.

This is a universally quantified expression consisting of an equality between an existentially quantified variable and an arbitrary term containing any of the universally quantified variables but not the existentially quantified one.

If MOR is already taking place on the goal, initiating further MOR is likely to be unwieldy or explosive, so a check is made that it does not currently contain any meta-variables. Specifically, since the generalised goal has the same form as the ungeneralised one, the lack of such a check would result in generalisations of generalisations, etc., in the case of the iterative deepening planner, although the depth first planner should never reach that depth in the list of methods again if it is successful.

Postconditions and Output

The generalisation is computed:

$$\forall x \in t_x \forall \vec{z} \in t_z \forall a \in t \exists y \in t.\ y =_t G\ of\ f(x, \vec{z})\ of\ a$$

The first output sequent is the proof of this generalisation from the original hypotheses. The second is the justification proof branch - that the generalisation implies the original conclusion:

1. $\forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.\ y =_t G\ of\ f(x, \vec{z})\ of\ a$

   $\vdash \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \exists y \in t.\ y =_t f(x, \vec{z})$

Tactic

The tactic, or macro of Oyster inference rules, is given here. The `seq` uses the cut rule to introduce the generalisation. After that, there are two branches, the first of which is the synthesis of the tail-recursive function. The second branch is the justification proof.

In the version of this method used by the supermethod, I do nothing more to these branches, leaving them to be completed by later methods. This is done by using `apply(idtac)`.

In the loosely-coupled non-supermethod version, the tactic below is supplied to complete the justification proof. I use the sequence of operations I have already described in the paragraph on justification in 7.4.2, interspersed with calls to the well-formedness tactic. The `intro` uses the generalisation hypothesis witness for the existential variable in the conclusion, and the `rewrite` finds a lemma which will complete the last simplification, of $f(x) = g(f(x), a)$, where a has been set to *nil*.

```
(seq(TR_Generalisation,new[gen])
 then [apply(idtac),
       repeat_intro_and_copy(gen,InstantiatedHyp) then
          elim(InstantiatedHyp,on(nil),new[je]) then
             [wfftacs,
              elim(je,new[jv,jw,jl]) then
                 intro(jv) then
                    [wfftacs,
                     rewrite(jw),
                     wfftacs
                    ]
             ]

      ]
 )
).
```

7.5.2 Longitudinal Wave Method

This method acts like the standard one, except that it takes account of the possibility that the input sequent could contain meta-variables. The unification of the left-hand-side of a wave rule with a subterm of the sequent's conclusion is controlled to allow higher-order unification, by using the wavefronts to identify the smallest and then successively larger subterms which must unify.

Given as input a sequent such as the generalisation:

$$\vdash \forall x \in t_x \forall \vec{z} \in t_z \forall a \in t \exists y \in t.y =_t G\ of\ (f(\boxed{c(\underline{x})}, \vec{z})\ of\ a$$

its functionality is described in this subsection.

Preconditions

I shall assume a longitudinal wave rule:

$$\phi(\boxed{\kappa(\underline{X})}, \vec{Z}) \rightarrow \boxed{\kappa'(\underline{\phi(X, \vec{Z})}, X, \vec{Z})}$$

Where upper case indicates variables, lower case constants, and a superscript arrow denotes the presence of zero or more arguments.

As usual, wave rules like this are selected and tested in turn. In standard CLAM the left-hand-side of the rule would be tested to see if it unified as a whole with some subterm of the sequent. The wavefront markers would automatically be aligned by any valid first order unification.

In the middle-out system, separate unifications are performed successively with each of the following, and progressive instantiation takes place.

1. the respective waveholes, x and X
2. the smallest terms containing the wavefronts, $c(x)$ and $\kappa(x)$
3. the whole left-hand side of the rule, $f(c(x), \vec{(z)})$ and $\phi(\kappa(x), \vec{Z})$. Note that if this ever matched something where f were variable, we'd have the problem of a flexible-flexible pair.

Although for the current task no higher-order meta-variables occur in the wavefront subterm, they might if this method were being used to reason middle-out about an induction, for example. Consequently, all unifications are performed higher-order. The unifications are performed with the wavefront markers removed, since it would be difficult to guess the type of these meta-functions.

Indeed it would be inappropriate, as they are not part of the type system. Since they still guide the terms chosen for unification, nothing is lost.

I am currently assuming a single occurrence of the induction variable. If there were several, there would consequently be several wave holes and wave fronts, and it would be necessary to try all the different combinations of rule wave fronts with conclusion wave fronts.

PostConditions, Output and Tactic

These are exactly as in the standard method, the rippled sequent is computed and planning continues. The tactic records the lemma to be applied, in which direction it is to be applied, and the position of the subterm to which it is to be applied. If the meta-variable were ever to become a complex term, this would be inadequate, I should have to store the position as partially variable too, and instantiate it as necessary.

7.5.3 Transverse Wave Method

This is like the standard transverse wave method, with the changes to permit meta-variables as described for the longitudinal wave method, but additionally, accumulators (more generally known as sinks, because of their rôle in relation to rippling) are unified before the whole left-hand-side is unified. This is essential to prevent unifications occurring between terms containing variables in adjacent positions such that flexible-flexible pairs would occur.

Given as input a sequent such as the generalisation:

$$\vdash \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \forall a \in t \exists y \in t.y =_t G \; of \; \boxed{f'(\underline{f(x,\vec{z})}, x, \vec{z})} \; of \; a$$

the method is as follows.

Preconditions

I shall assume a transverse wave rule:

$$\phi(\boxed{\kappa(\underline{X})}, \vec{Z}, A) \rightarrow \phi(X, \vec{Z}, \boxed{\kappa'(\underline{A})})$$

The general form of a transverse wave rule allows for multiple wavefronts on the left hand side and multiple sinks. I have assumed one of each, to avoid the problem of deciding which to match with which in the meta-sequent's conclusion and the rule, and since it is unusual for there to be more. As for the case of the longitudinal rule, in general, all possible pairings would have to be considered.

The positions of the wavefronts before and *after* the use of a lemma are used to ensure that the effect of the lemma is a transverse ripple as intended.

Sinks are determined by induction hypothesis quantification, not rule syntax. Essentially everything proceeds as for the longitudinal wave method, except that there is an extra unification stage 2', in which we insist that the sink selected by the method is the accumulator added by the generalisation.

As I have already described in chapter 4, any unification which binds a universally quantified variable into the identity of a function variable is rejected.

PostConditions, Output and Tactic

Again, these are just the same as in standard CLAM

7.5.4 Existential Method

This is very similar to the standard existential method. It has been extended so that it automatically introduces any universally quantified variables in the conclusion as free variables. It has been restricted so as not to apply to goals already containing meta-variables.

This method and the tail-recursive generalisation one unfortunately apply in almost indistinguishable circumstances. However, if this were to be applied to the initial goal:

$$\vdash \forall x \in t_x \forall \vec{z} \in t_{\vec{z}} \exists y \in t.y =_t f(x, \vec{z})$$

it would simply use $reverse(x)$ for y, and the proof would end trivially. To avoid these trivial completions, the tail-recursive generalisation method precedes the existential one in the list of methods, which is appropriate, since the former is a special case of the latter.

A further problem with this method was that it could cause useless search in the iterative deepening planner by suggesting meta-variable solutions before other methods such as symbolic evaluation had been applied. The only way of stopping this was to insert artificial restrictions requiring that in an equality of the form $Y = term$, say, where *term* was instantiated, that *term* be atomic. This was another pointer to the need for a supermethod.

7.5.5 Symbolic Evaluation Method

Some of the rewritings done by the symbolic evaluation method, sym_eval, achieve similar functions to the wave method, because they also rewrite terms according to function definitions. These were adapted correspondingly to the wave rule adaptations.

7.5.6 Order of Considering Methods

The following table gives the order of consideration of methods. The tail-recursive supermethod, (tr_gen_strat), may or may not be present.

Standard	MOR
tautology	tautology
sym_eval	sym_eval
wave	casesplit
casesplit	strong_fertilize
strong_fertilize	weak_fertilize
weak_fertilize	generalise
generalise	wave
existential	(tr_gen_strat)
induction	tr_gen
ind_strat	existential
	induction
	ind_strat

The new method introducing a tail-recursion generalisation is low in the list because it only applies rarely, in quite specific circumstances. Placing the new method higher would be inefficient, because it would be considered frequently and fail to be applicable. It must come before induction so as to be chosen preferentially, since induction will always apply whenever the new method does. The tail-recursive generalisation method must also precede the existential method, since it handles a special case for which the latter method would be inappropriate.

The wave method has been moved below the fertilization methods, because it is now expensive to try, with higher-order unification built in.

In the system with the tail-recursive generalisation supermethod, it takes precedence over the method which just introduces a generalised formula.

7.6 MOR on the *reverse* example

I shall describe the proof from the point of view of the depth-first and iterative-deepening planning processes and explain how MOR takes place. At each point, I will describe the choices available, and show how the system discriminates.

At the start, the theorem is

$$\vdash \forall x \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$$

The only methods whose preconditions may be satisfied are induction and tail-recursive generalisation. The latter precedes induction in the list of methods, so it is chosen. The result is a new generalisation meta-sequent:

$$\vdash \forall x \in pnat\ list \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} G(reverse(x), a) \quad (7.7)$$

and a further goal to show that proving this justifies the original goal. I will return to this justification branch later.

7.6.1 Proving the Generalisation Using Induction

The only methods which might conceivably apply to 7.7 are existential, tail-recursive generalisation and induction. The first two are barred since the meta-sequent already contains a meta-variable. Induction is unaffected by the meta-variable. It finds that the only universally quantified variable is x, and that occurs in *reverse* in an argument position affected by a wave rule. A straightforward list induction is suggested, based on the scheme indicated by the wave rule.

The Step Case

1. $h \in pnat$

2. $t \in pnat\ list$

3. $\forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} G(reverse(t), a)$

$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} G(reverse(\boxed{h :: \underline{t}}), a)$

The only applicable method for this meta-sequent is the wave method. The existential and tail-recursive generalisation's preconditions would fail due to the presence of the meta-variable. The longitudinal version of the wave method is applied in preference to the transverse version, because it appears first in the list of methods.

The wave method could select several wave rules, depending on which were available. It is always debatable, when discussing wave rules, which ones should be assumed to be available. There are broadly four options:

- Minimal - Only necessary definitions, anything else to be created as required.
- Average - Definitions, along with some reasonable collection of lemmas. Specifically *not* just such lemmas as will make life easy for the proof in hand.
- So Far - Everything you happen to have proved so far.
- Maximal - everything you can think of, no matter how trivial, which is true for the theory.

The last of these is impossible. The next-to-last is relative to which day you run the system on, and not different enough from what I have labelled the "average" case to be useful. The first is an interesting but not particularly realistic case. What mathematician or automatic theorem prover would we expect to derive everything from first principles all the time? I will present an average problem, and say what lemmas are available. By explaining what choices they present to the system reasoning middle-out, it should be clear that they do not make the task artificially simple.

The advantage of using planning is that the methods will work as best they can with whatever information is available about the theory and its theorems.

A typical collection of wave rules would be those in tables 7–1 and 7–2, containing the longitudinal rules and the transverse ones respectively. The order in which the rules are found by the system depends on the order in which they were loaded. One would expect the system either to be impervious to order, or to select an order to suit. In fact it does a combination of the two. Since the longitudinal version of the wave method is considered first, this filters the rules to get the longitudinal ones, so effectively they always precede the transverse ones. Beyond that, the system takes rules as they come. The ordering of the rules was selected so that the "obvious" rules weren't conveniently first. In fact there is minimal search. Usually only one induction scheme applies, and at the worst two wave rules will apply in different ways, usually leading to different, but valid, solutions. Problems only arise when both a longitudinal rule and a transverse rule can apply, and due to the ordering, the system wrongly attempts the longitudinal.

The method tries to apply a longitudinal rule, for preference. The only possible match is the definition of reverse. This applies, producing a new sequent:

1. $h \in pnat$
2. $t \in pnat\ list$
3. $\forall a \in pnat\ list \exists y \in pnat\ list. y =_{pnat\ list} G\ of\ reverse(t)\ a$

$\vdash \forall a \in pnat\ list$

$$\exists y \in pnat\ list.\ y =_{pnat\ list} G\ of\ \boxed{append(\underline{reverse(t)}, h :: nil)}\ of\ a$$

This is submitted afresh to the planner, and exactly the same methods apply as for its predecessor - the wave method. Again, longitudinal waves are considered first. Since this unification results in potentially explosive flexible-flexible unifications, at first I tried barring such matches at this point, but it was difficult to do in a principled way. Eventually I used the iterative-deepening planning which will allow this match but also considers the transverse rules at the same

time, and come up with a viable answer. This problem reinforces the lesson of other parts of this experiment in controlling MOR, that knowledge about the structure of the task should be used to the full, for in a supermethod, we could describe those sorts of combinations of rippling which were acceptable.

Each transverse wave rule is considered in turn. For each, as described before, there is progressive unification of the terms within the wavefront, the wavefront term, and the whole left-hand-side of the rule with whatever subterm of the conclusion it will match. At any point, one of these may fail, and the planner will backtrack to get the next wave rule.

Here either 7.28 or 7.30 could apply, and the former is chosen purely because it is encountered first. Consequently, G is identified as $\lambda u \lambda v.append(u, v)$ and the rule is applied.

1. $h \in pnat$
2. $t \in pnat\ list$
3. $\forall a \in pnat\ list$

$$\exists y \in pnat\ list.y =_{pnat\ list} \lambda u \lambda v.append(u, v)\ of\ reverse(t)\ of\ a$$

$$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(t), \boxed{h :: \underline{a}})$$

since the conclusion part is β-reduced.

Lastly, in this branch, strong fertilisation applies. It still identifies the induction hypothesis, because it applies β-reduction to it.

The Base Case

The step case work has instantiated G, so the sequent is

$$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(nil), a)$$

The existential method could apply, and would introduce

$$append(reverse(nil), a)$$

for y. This would be cumbersome, but not wrong. Symbolic evaluation occurs early in the list precisely to avoid this kind of thing. It is the first method to catch this. It applies twice, using the base definitions of *reverse* and *append*:

$$\vdash \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} a$$

Now, the existential method is the only one which can apply, it introduces a and a meta-variable Y for y:

1. $a \in pnat\ list$

   $\vdash Y =_{pnat\ list} a$

Lastly, the tautology method applies, instantiating Y to a, and completing the meta-proof branch. No other method could apply here.

7.6.2 The Justification

1. $\forall x \in pnat\ list \forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(x), a)$

   $\vdash \forall x \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$

By now, the variable function has been instantiated. The sequent is proved as follows:

First, CLAM assumes that any universally quantified variables in the conclusion should be identified with their counterparts in the generalisation hypothesis. So it introduces each of these, renaming it, and echoing this for the hypothesis.

2. $x : pnat\ list$
3. $\forall a \in pnat\ list \exists y \in pnat\ list.y =_{pnat\ list} append(reverse(x), a)$

   $\vdash \exists y \in pnat\ list.y =_{pnat\ list} reverse(x)$

An a must be chosen such that the y which is then yielded by the generalised hypothesis will indeed be a suitable value as the conclusion y, i.e.

$$append(reverse(x), A) =_{pnat\ list} reverse(x) \tag{7.39}$$

CLAM uses a submethod which is the hypothesis equivalent of the existential method, and, naturally, it works on universal quantifiers. A meta-variable is inserted for a, and the symbolic evaluation method, which includes both the base submethod and the reduction submethod, is applied to that hypothesis alone, i.e. to the expression $\exists y \in pnat\ list.y =_{pnat\ list} append(reverse(x), A)$ to arrive at an expression to eliminate a on.

7.6.3 Using a Supermethod

Combining the knowledge of the overall structure into a supermethod sorted out the problems I have described in trying to drive and restrain methods intended for particular situations in the absence of knowing whether their situation obtained or not.

The supermethod follows the proof structure I have described in this chapter. The supermethod's code is in appendix A, and it is described now. In the account that follows, *applicability* is used in a technical sense to mean that CLAM attempted to use the method or submethod, and ensured that its preconditions and postconditions would succeed.

The supermethod's preconditions are the applicability of the existing tail-recursive generalisation method, and then the applicability of the induction method to the result. With the supermethod, all the methods selected as applicable lead to successful solutions.

The postconditions deal with the step case(s) and base case(s) separately. A new *ripple-over* submethod was written to apply at least one longitudinal wave followed by a transverse one. It looks for a solution using a minimal number of longitudinal waves. This was used for the step case(s), followed by a fertilization. The base case(s) are as before, except that now the existential method can be used without needing restrictions, because it is being used where it is needed.

The justification part identifies the initialisation value for the accumulator by MOR, as described earlier, It then constructs a tactic virtually identical to the one described for the free-standing tail-recursive generalisation tactic.

The only real difference is that the initialisation value has now been supplied automatically.

7.7 Results

This approach is also successful in synthesising a tail-recursive version from a naïve one for *times*, *total*, *greatest*, *multhalf*, summation of the values of a function from 0 to an arbitrary value, and *length*. The theorems, definitions and instantiations for G (assuming much the same collection of lemmas and therefore wave rules as above) are listed in appendix B.

7.8 Conclusions

This chapter has given an account of extensions to the CLAM proof planning system to detect goals which describe the naïve specification of a program, and create synthesis proofs completing such goals which yield tail-recursive algorithms. The technique implemented in these extensions relies on the programs-as-proofs philosophy to characterise proof structures which ensure that the corresponding programs will be tail-recursive. Achieving such proof structures requires adaptation of the existing specification, to specify a new tail-recursive function, and two accompanying proofs, one being the synthesis of the new function, and the other the justification that the new function supplies the same values as the original. The extensions to CLAM have to deal with the problem of not knowing initially what the new function must be. They do this by using higher-order meta-variables to describe the new specification, and then allowing the subsequent proof to instantiate the meta-variables, using MOR. The tail-recursion proof structure is used to ensure the proof corresponds to a tail-recursive function.

The extensions fit well into the existing CLAM planning structure, since they are inductive proofs. Only a few methods needed to be created or modified,

although the use of higher-order unification required careful interfacing. *MOR* itself made special demands on the planner, because it became important to pursue first those proof branches which would decide the identity of the meta-variables rapidly and conclusively. The subsidiary goal of justifying the choice of the new function also relied on *MOR*, to fix the initial value of the accumulator.

Attempts to extend CLAM's use of methods in a freestanding way were of limited success, and were eventually superseded by a supermethod, incorporating available knowledge of the proof structure to direct appropriate methods.

$$length([X :: \underline{Y}]) \rightarrow [\underline{s(length(Y))}] \tag{7.8}$$

$$[X * \underline{Y}] + [X * \underline{Z}] \rightarrow [X * \underline{Y + Z}] \tag{7.9}$$

$$X * ([\underline{Y + Z}]) \rightarrow [\underline{X * Y} + X * Z] \tag{7.10}$$

$$X * ([Y + \underline{Z}]) \rightarrow [X * Y + \underline{X * Z}] \tag{7.11}$$

$$append([append(X, \underline{Y})], Z) \rightarrow [append(X, \underline{append(Y, Z)})] \tag{7.12}$$

$$append(X, [append(\underline{Y}, Z)]) \rightarrow [\underline{append(append(X, Y), Z)}] \tag{7.13}$$

$$([X * \underline{Y}]) * Z \rightarrow [X * \underline{(Y * Z)}] \tag{7.14}$$

$$X * [(\underline{Y} * Z)] \rightarrow [\underline{(X * Y)} * Z] \tag{7.15}$$

$$([X + \underline{Y}]) + Z \rightarrow [X + \underline{(Y + Z)}] \tag{7.16}$$

$$X + [(\underline{Y} + Z)] \rightarrow [\underline{(X + Y)} + Z] \tag{7.17}$$

$$max(X, [max(\underline{Y}, Z)]) \rightarrow [\underline{max(max(X, Y), Z)}] \tag{7.18}$$

$$max([max(X, \underline{Y})], Z) \rightarrow [\underline{max(X, max(Y, Z))}] \tag{7.19}$$

$$[s(\underline{X})] * Y \rightarrow [\underline{X * Y} + Y] \tag{7.20}$$

$$greatest([X :: \underline{Y}]) \rightarrow [\underline{max(X, greatest(Y))}] \tag{7.21}$$

$$max([s(\underline{X})], [s(\underline{Y})]) \rightarrow [\underline{s(max(X, Y))}] \tag{7.22}$$

$$total([X :: \underline{Y}]) \rightarrow [X + \underline{total(Y)}] \tag{7.23}$$

$$[s(\underline{X})] + Y \rightarrow [\underline{s(X + Y)}] \tag{7.24}$$

$$reverse([X :: \underline{Y}]) \rightarrow [\underline{append(reverse(Y), X :: nil)}] \tag{7.25}$$

$$append([X :: \underline{Y}], Z) \rightarrow [X :: \underline{append(Y, Z)}] \tag{7.26}$$

Table 7–1: Longitudinal Rules

$$append(\boxed{append(\underline{X}, Y :: nil)}, Z) \rightarrow append(X, \boxed{Y :: \underline{Z}}) \quad (7.27)$$

$$append(X, \boxed{Y :: \underline{Z}}) \rightarrow append(\boxed{append(\underline{X}, Y :: nil)}, Z) \quad (7.28)$$

$$append(\boxed{append(\underline{X}, Y)}, Z) \rightarrow append(X, \boxed{append(Y, \underline{Z})}) \quad (7.29)$$

$$append(X, \boxed{append(Y, \underline{Z})}) \rightarrow append(\boxed{append(\underline{X}, Y)}, Z) \quad (7.30)$$

$$\boxed{(\underline{X} * Y)} * Z \rightarrow X * \boxed{(Y * \underline{Z})} \quad (7.31)$$

$$X * (\boxed{Y * \underline{Z}}) \rightarrow \boxed{(\underline{X} * Y)} * Z \quad (7.32)$$

$$\boxed{(\underline{X} + Y)} + Z \rightarrow X + (\boxed{Y + \underline{Z}}) \quad (7.33)$$

$$X + (\boxed{Y + \underline{Z}}) \rightarrow \boxed{(\underline{X} + Y)} + Z \quad (7.34)$$

$$X + \boxed{s(\underline{Y})} \rightarrow \boxed{s(\underline{X})} + Y \quad (7.35)$$

$$\boxed{s(\underline{X})} + Y \rightarrow X + \boxed{s(\underline{Y})} \quad (7.36)$$

$$max(X, \boxed{max(Y, \underline{Z})}) \rightarrow max(\boxed{max(\underline{X}, Y)}, Z) \quad (7.37)$$

$$max(\boxed{max(\underline{X}, Y)}, Z) \rightarrow max(X, \boxed{max(Y, \underline{Z})}) \quad (7.38)$$

Table 7–2: Transverse Rules

Chapter 8

Comparison with Related Work – Tail Recursion Optimisation

In attempting any comparison it is important to remember that although the underlying problems attempted by different researchers may be similar, their overall purpose can be quite different, and this will naturally influence the emphasis of their work.

One of the most significant differences between the work described in this thesis and other work on tail-recursion optimisation is that the approach taken in my work exploits the proofs-as-programs paradigm. This impacts on how the equivalence of a program to its specification is verified, and how we create programs having desired properties. The guarantees for both of these lie in the form of the proof used. Outwith the proofs-as-programs setting, such guarantees must be sought through proofs about the process creating the program. In proofs-as-programs, equivalence of a specification to a program is part of the proof relating to that particular program. Elsewhere, preservation of equivalence is a general property that must be proved of program synthesis or transformation system over all operations it could perform. In proofs-as-programs, properties of programs become properties of proofs.

The proof-theoretic properties (in constructive logic) corresponding to tail-recursion in their corresponding programs can be characterised [Wainer 89]. The work described in this thesis is a first attempt to automate the generation of

proofs meeting these characterisations, starting from a naïvely recursive specification.

Constraining the synthesis proofs to using certain proof structures ensures that the corresponding functional programs are tail-recursive. The problem becomes one of producing these proof structures – a task well-suited to a proof-planning system. Some questions arise:

- is the proof structure used here sufficient to cover all tail-recursive optimisations?
- are the chosen representation and its implementation adequate to express all the variants of the proof form which I have described in chapter 7 as characterising tail-recursiveness?
- how will the planner produce such proofs?

The MOR proofs-as-programs approach is goal-driven in the sense that it starts from the intention of creating tail-recursion and works back towards the tail-recursive form, with the aid of the original program as specification. Any extra constraints will be proved as they are needed. The equivalence of the specification and result is a requirement of the proof process.

Key work on generating tail-recursive programs has come from Darlington and Burstall [Darlington 81, Burstall & Darlington 77], which will sometimes be referred to as *fold-unfold*, for convenience. By categorising program transformation operations according to their function, they were able to assemble combinations of transformations automatically. By placing demands on the combination patterns, particular effects could be achieved. This is analogous to proof planning. One of their combinations would take a defined program (or set of recursion equations) and produce a tail-recursive equivalent. Each transformation operation is guaranteed to preserve the equivalence of the original and the result of its transformation, so verification of equivalence of the starting program and the final result was not an issue considered at length. The significant decisions were the definition of the original task, what information was given to the system,

and the nature of the automatic control. I shall go into this in more detail in section 8.1.

An alternative approach driving previous work in this area has taken programs with known tail-recursive formulations and generalised the observed patterns linking them into schemas. This is the approach taken in Cooper's paper [Cooper 66], which is generally cited as one of the first in this area. Each generalised schema has then necessarily been restricted to ensure that when it is applied to suggest an alternative version of some function, the new expression produces the same results as the original. The constraints implied by these restrictions arise from the schema-level proofs of equivalence between input and output schemas. The restriction takes the form of theorems about the entities named in the schema, such as that some function must be associative, or be the inverse of another. The guarantee that the aim of the optimisation has indeed been achieved is implicit in the structure of the output schema. Template[1]-based approaches are more data-driven, in that a number of templates could apply, perhaps designed to achieve different kinds of optimisation. The major contributions to the template approach come from Darlington [Darlington 72], who automated the application of some of these transformations, and from Huet & Lang [Huet & Lang 78], who extended the library of templates, and developed proofs of their validity, but without any automation.

In [Huet & Lang 78] Huet & Lang provide a thorough theoretical account of the validation of second-order schema and constraint templates. They give their concerns as being:

- How to discover templates;

---

[1]I am deviating from Darlington's nomenclature here for the sake of consistency - he uses the term *schema* for what Huet & Lang term a *template*. I shall adopt Huet & Lang's terminology, and use *template* to encompass a set of input pattern, output pattern plus associated constraints, reserving *schema* for the input or output patterns within templates.

- How to validate them as preserving the equivalence of programs;
- How to recognise that a template is applicable to a given program;
- How to organise a system automatically applying such templates.

The first of these is not tackled in the paper, since as they point out, "it is one of the basic problems of communicating knowledge about programming", and they wish to concentrate on the second and third. The fourth is not addressed explicitly, but mentioned in passing from time to time, usually as a fruitful direction for future research. Since they are not describing an implementation, issues of correctness are more important to them than automation and search control.

It is apparent, but not surprising, that the criteria relevant to the proofs-as-programs approach are not the same as those stated for the schema approach. I shall return to both sets of criteria at the end of this chapter, after examining the differences between these approaches in detail. Before going on to that I will set an agenda, in approximate order of centrality, by noting factors to be considered:

1. Use of a proof-based criterion for tail-recursiveness. This provides us with a way of characterising our goal very generally, something the other approaches lack. As long as the proof satisfies the requirements of the tail-recursive characterisation, the program will be tail-recursive. The template approaches are driven by finding existing transformations and generalising their patterns into templates.

2. Extent of automation versus human intervention. The MOR/CLAM system is entirely automated, where other systems need a degree of human guidance, or are not automated at all.

3. Search problems and search control. CLAM's proof-planning information is well-suited to the task and supplies exactly the kind of information

needed. It provides a principled framework for describing the progress of these proofs and explaining their properties.

4. Equivalence of schemas versus equivalence of programs:

   - Templates are relatively static, their only features which can be selected dynamically, in response to a given task, are the choices which may be made for their variable entities. They are consequently constrained by the need to anticipate all the possibilities in advance. This is exacerbated by the need to prove equivalence *of the schemas* in advance, too, demanding further restrictions, sufficient to cover all the cases described by the template. However, the proof need only be done once for each template.

     When programs are built dynamically, there is the potential to select components according to the demands of the individual case. For proofs-as-programs, equivalence is more straightforward, as only such equivalence as relates to the case in hand need be proved. Such a proof must be supplied for each synthesis.

   - Choice of intermediate level - second-order schemas, control of program transformation operations such as folding and unfolding, or control of proof. Each possibility constrains the nature of description possible. A schema describes an input, an output and constraints, but nothing about *how* to get from input to output, short of what is implicit in unification. Program transformations describe a pattern of operations on program fragments, but without an overall statement about how the end product will relate to the input - typically there is no formal justification of the whole transformation as opposed to each step. Proof control permits both a process and an input-output relationship to be described.

   - Nature of justification of equivalence. The proof control approach used in this thesis requires only an auxiliary proof branch specific to the case in hand, and although a general method must be pro-

duced for finding such proofs, that has been found to be relatively straightforward. The program transformation approach requires a higher-level proof assuring the equivalence-preserving properties of each transformation operation. Similarly, the generalised template approach requires a higher-level proof for each template that the general transformation it describes is equivalence preserving provided that the (general) constraints hold.

- Nature of assurance that optimisation is actually taking place. This is a property implicitly designed into the output schema of each template Huet & Lang give, rather than being proved or designed into a (non-existant) template producing process. Optimisation is not assured for the fold-unfold system. Achieving the required proof structure guarantees tail-recursiveness under proofs-as-programs.
- Different information is made explicit or kept implicit in the various systems. In proofs-as-programs and fold-unfold, objects' rôles and relationships may be implicit. Depending on the implementation, the overall structure ensuring tail-recursive proof may be implicitly imposed, or explicitly checked. Templates are necessarily explicit about how entities are involved, but not why.

5. Type of unification and/or matching;

6. Explicit use of meta-variables to enable argument at the meta-level. The template approaches have meta-variables, but there is no attempt to use them for explicit guidance, it is bundled into the constraints which are placed on them.

7. What are the functions of different templates? Do they amount to consideration of possible correspondence to different rippling patterns, where wavefronts move up and then across into an accumulator, or up and then down a different branch, or perhaps some different configuration?

8. Use of other known theorems;

9. Range of accessible problems – different recursive structures and beyond second-order;

10. Relation to known processes for linear recursion;

11. Completeness and termination;

12. Transparency of the resulting program;

13. Explainability of the process;

14. Choosing between alternative solutions;

15. Effect of using a proof development system;

16. Other types of optimisation;

17. Evaluation system - call by name/call by value

18. Closeness to any actual language or implementation;

19. Efficiency of the synthesising operation;

20. Modifiability and extensibility.

To aid my comparison, in 8.1 - 8.3, I will run through the example I used in chapter 7 – the optimisation from naïve to tail-recursive *reverse* – both from the different perspectives of Darlington and from that of Huet & Lang. Relating to this concrete example will make it easier to describe some features, and point up differences between the four main approaches I will contrast.

8.1 Burstall and Darlington's Fold-Unfold Approach

I shall work through an example first as illustration, and give an account of the fold-unfold method afterwards.

The specification of the *reverse* function, and that of <> (append) which it relies on are given to the system:

$$reverse(nil) == nil \tag{8.1}$$

$$reverse(h :: t) == reverse(t) <> h :: nil \tag{8.2}$$

$$nil <> l == l \tag{8.3}$$

$$(h :: t) <> l == h :: (t <> l) \tag{8.4}$$

along with the knowledge that <> is associative ($h, l.t, u$ and x are all variables). Other lemmas may also be provided for use as rewrite rules, but as equations. Only those with well-known properties, and which can loop readily are classified as laws. Although some of this terminology has now entered common usage, definitions are still in order:

Unfold. Substitute for an expression the result of evaluating its outermost function by using the function's definition once, e.g. 8.2 from left to right.

Fold. The opposite of unfold, substitutes the definitional version for the expanded one, e.g. 8.2 from right to left.

Laws. Knowledge of the commutativity and associativity of functions in the current problem.

A "eureka" step is given to start the system, a new function, $reverse_{tr}$ is defined by:

$$reverse_{tr}(x, u) == reverse(x) <> u \tag{8.5}$$

Recursion is assumed to be defined on x. Now, the automatic part of the system can examine the cases which would have to be defined. First the base case, is found by instantiating the definition of $reverse_{tr}$ (8.5), and unfolding using the definitions of *reverse* and <>:

$$\begin{aligned} reverse_{tr}(nil, u) &== reverse(nil) <> u \\ &== nil <> u \\ &== u \end{aligned}$$

The step case's definition is unravelled similarly, but an extra step uses the associativity of <> to bring the $reverse_{tr}$ definition into a form which can be folded with the original definition (8.5):

$$\begin{aligned} reverse_{tr}(a :: x, u) &== reverse(a :: x) <> u && (8.6) \\ &== (reverse(x) <> (a :: nil)) <> u && (8.7) \\ &== reverse(x) <> ((a :: nil) <> u) && (8.8) \\ &== reverse_{tr}(x, (a :: nil) <> u) && (8.9) \end{aligned}$$

Finally, the circle is completed, and *reverse* can be defined in terms of $reverse_{tr}$, by folding 8.2 with 8.5:

$$reverse(a :: x) == reverse_{tr}(x, a :: nil)$$

I shall not describe the system in total, just the portion which is relevant to my comparison here.

The instantiation stage can be given some assistance from the user to set the system off working on the right equations. Then the automatic part takes over. It applies the transformation operations - folds, unfolds and uses of other lemmas and laws of the theory - according to one of a number of strategies.

Their design of strategies was guided by their observations that:

- Almost all the optimising transformations consist of a sequence of unfoldings and rewritings by lemmas and then foldings.

- Use of associativity and commutativity can usually be delayed until just before folding.

The latter observation led them to the specialised notion of a *forced fold.* This restricted use of laws (associativity or commutativity) to such applications as made a fold possible, and so cut down on pointless use.

An exhaustive (with backtracking) but relatively inefficient strategy was:

Algorithm 1:

1. Arbitrarily do an unfold or rewriting by a lemma. Repeat this or proceed to stage 2.

2. Do an arbitrary forced fold. Repeat until no more folding is possible.

The further observation that in most cases folding could be delayed until until all possible unfolding had been done, provided that the equations were kept in unfolded form lead to the following less general, but more efficient algorithm:

Algorithm 2:

0. Unfold each equation until no further unfolding is possible.

   Then, for each instantiation case of the equation to be improved:

1. Unfold until no further unfolding is possible;

2. Arbitrarily either do a rewriting by a lemma and go to stage 1, or go to stage 3;

3. Do an arbitrary forced fold. Repeat this stage until no more folding is possible.

The overall goal was to get the new equations to such a form that folding became possible, giving the strategy purpose. This system was able to improve a range of examples.

The system described in [Darlington 81] refined these strategies further, mainly to achieve better control over the development. New areas were attempted, such as generating other improved functions which also optimised using an accumulator. An example of this being the versions of functions which start from a base value, and build the calculation up from the bottom, i.e. recursion is converted into a "going up" form from a "going down" one.

8.1.1 Parallels with *MOR* System

Parallels between the types of components should be immediately apparent. This is no accident, as the proof plans work grew out of the recognition that certain theorem proving operations, such as the use of wave rules, could be categorised similarly to the fold and unfold operations. However, it would be misleading to stretch this analogy too far, as divergence has taken place since.

The initial specification of the new function, (8.5), to be elaborated in tail-recursive form, achieves the same kind of function as an induction - it suggests the induction variable, and provides the equivalent of an induction hypothesis for folding to aim for. The equivalent of an induction's cases is handled by using the constructors for the type of the variable, so only recursive schemes corresponding to those are permitted. The function which the *MOR* system finds during the proof process is given by the definition at this point.

Burstall and Darlington's unfold operations only relate to definitions, so they correspond to CLAM's base and step submethods, and to such wave rules as apply definitions. Their folding is not the same as rippling down into sinks. Folding includes their equivalent to fertilisation, since it recognises that an expression compatible with the posited new definition has been reached, and terminates the improvement. If repeated, it also incorporates a recent CLAM technique of reversing rippling after weak fertilisation, to improve the chance of a cancellation. Associativity and commutativity, are treated specially, and only used to enable folding. This contrasts with the CLAM approach, where lemmas are preprocessed to establish their usages as wave rules or simplifications, and used according

as they fulfill meta-level requirements. Using other lemmas in Burstall and Darlington's system is controlled by orienting them as rewrite rules in advance.

The overall pattern of applications of categories of program transformations is, not surprisingly, similar to those performed in proofs.

8.1.2 Conclusions

Burstall and Darlington's system needed no higher-order unification, and there is no reason why it should have any problem with higher-order problems than have been described in their papers. It was able to work in conjunction with other kinds of program improvement, something that has not yet been tried for the proofs-as-programs approach.

Less of the Burstall and Darlington system has been automated (though that was not a goal of Darlington's transformation development system). Fewer recursive schemes are available.

The treatment of non-definitional lemmas of whatever kind is not controlled in such a principled way as in CLAM, and the lack of any equivalent of wavefronts to direct and monitor progress means that their strategy is less controlled.

The point at which the associativity is used to enable the terminating fold in the *reverse* proof above, (8.8) is a key step, analogous to CLAM's transverse rippling. In CLAM, any suitable transverse wave could be used, whereas here, only associativity (or a combination of associativity and commutativity) of the function which had been inserted as the definition would be allowed. Indeed later versions of CLAM would be able to assemble the effect of a transverse wave from appropriate longitudinal ripples, and then carefully controlled backwards ripples into a sink.

The Burstall and Darlington system doesn't have as much of a search problem at the stage when the associativity is used (8.8), exactly because they constrain the possibilities.

There is no guarantee that Burstall and Darlington's application of the transformations will improve the program. The fold-unfold systems attempt a strategy of performing operations which complete the definition of a new function. There is nothing inherent to the strategy of application or the conditions for termination which ensure that optimisation will necessarily have taken place. Features with the capability to optimise have probably been installed, but there is no characterisation enforcing an optimised structure or monitoring the result.

8.2 Darlington's Template System

This is part of Darlington's thesis work, already described briefly in chapter 2. It automates the use of tail-recursion producing template-driven transformations.

I will take a naïve definition of *reverse* as a starting point

$$\begin{aligned} reverse(l) \Leftarrow \underline{if}\ null(l)\ \ \underline{then}\ \ nil \\ \underline{else}\ \ reverse(cdr(l)) <> cons(car(l), nil) \end{aligned} \quad (8.10)$$

This is equivalent to a set of recursion equations which exhaustively and mutually exclusively define the *reverse* function.

8.2.1 Schema Matching

The first task is to see if any of the schemas from the templates describing recursion optimisation translations, fit. Darlington's system would proceed automatically, checking through schemas, using F-matching to detect a match. His first schema template would fit. The input pattern (slightly adapted) is

$$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ \underline{if}\ c(x)\ \underline{then}\ h(d(x), f(e(x))) \quad (8.11)$$

All of a, b, c, d, e, f, g and h are variables of the appropriate types, number, list, function etc. Since the schema must represent a set of mutually exclusive recursion equations, $a(x)$ and $c(x)$ must be exhaustive and mutually exclusive, i.e.

in this case $c(x) \Leftrightarrow \neg a(x)$. Output is an iterative POP-2 program, which I will present here in an ALGOL-like form:

```
if a(x) then ans := b(x)
     else [ ans := d(x);
            x := e(x);
            while c(x) do [ans := h(ans, d(x));
                           x := e(x)
                          ];
            ans := h(ans, b(x))
          ]
```

There are attached conditions, that:

1. whatever h is instantiated to does not contain as a subexpression whatever x is instantiated to. This is a restriction on the solutions suggested by the matcher, equivalent to the one I use to filter out unsuitable unifiers. It ensures that the functions, which are supposed to operate on x, don't include explicit references to x by name;

2. the function that h is instantiated to is associative.

Provided a substitution can be found which satisfies these conditions and gives values for all the variables, the output part of the schema template uses them to supply an iterative program. In this case, matching (8.10) to (8.11) the following substitutions would be found, ignoring any which fail the first condition. Their values for some variables are identical to each other:

x	a	b	c	e	f
l	$\lambda u.null(u)$	nil	$\lambda u.\neg null(u)$	$\lambda u.cdr(u)$	$\lambda u.reverse(u)$

apart from their alternative substitutions for d and h:

	d	h
1.	$\lambda u.cons(car(u), nil)$	$\lambda u \lambda v.v <> u$
2.	$\lambda u.car(u)$	$\lambda u \lambda v.v <> cons(u, nil)$
3.	$\lambda u.u$	$\lambda u \lambda v.v <> cons(car(u), nil)$

Note in passing that the substitution for d in the second of these is not of the right type to fit in with the type that h would require in that substitution. Although c is not supplied directly by the instantiation, its computation is straightforward.

With these substitutions the second condition is:

	$h(u, h(v, w)) = h(h(u, v), w)$
1.	$(w <> v) <> u = w <> (v <> u)$
2.	$(w <> cons(v, nil)) <> cons(u, nil)$
	$= w <> cons(v <> cons(u, nil), nil)$
3.	$(w <> cons(car(v), nil)) <> cons(car(u), nil)$
	$= w <> cons(car(v <> cons(car(u), nil)), nil)$

The condition arising from the first of these is satisfied, since it is recorded as a basic property of $<>$. An algebraic manipulator is used to check such conditions, using a mixture of recorded properties of the theory and user interaction, depending on circumstances, as described next. The other three substitutions' conditions fail to be established. They are variously ill-formed or not true.

8.2.2 Using Properties of a Theory

Basic properties (such as commutativity and associativity) of functions used in the definitions are supplied to the system and used by its algebraic manipulation routine to test conditions. The algebraic manipulation routine is a rewriting system, which, given a set of equivalent formulae (rules) and an expression, produces the set of all the expressions equivalent to the first under one rewriting

using the rules. A simple breadth first search with loop-checking was enough to test the truth of any conditions likely to be encountered. This was further controlled by placing resource limitations on the manipulator and taking it to have failed if no success was forthcoming within the resources allowed. It was not a major part of Darlington's system, but was adequate for his purposes.

In this example, the conditions can be checked automatically. As noted in chapter 2, for other schema the user might be asked to supply an inverse for a function, or a unique value satisfying a predicate.

8.2.3 Search Strategy

Darlington's search strategy was to use backtracking. Each template consisted of a single input pattern which could turn into any of a number of output programs depending on whether the associated conditions for each output were satisfied. The system would pick a template and find a substitution. The template would supply a number of possible output programs with associated conditions which would have to be satisfied for the use of the template to be valid. If on testing the conditions any failed, it would backtrack over the other output forms testing their conditions until it found a success. If none of those succeeded, backtracking would find another substitution and try all the outputs again. Only when all substitutions had been attempted against all possible outputs' conditions and failed would it give up on a particular template, and backtrack to the next.

8.2.4 Validation of Schemas

The emphasis of this system was on achieving automatic optimisations, so known templates were supplied to this system and not derived automatically. Their validation was assumed to have been a pre-processing step.

8.3 Huet & Lang's Template System

Since they are not describing an automated system, Huet & Lang simply state that a particular template would be applicable, and give the preferred substitution set. They discuss the efficiency and search aspects of their approach. For them search exists amongst templates and results from the matching algorithm.

The library of templates given in [Huet & Lang 78] is collected from a variety of sources, particularly Darlington's thesis. They have distilled some of the schemas into more general forms. This has some drawbacks, which I shall return to.

In this account I shall follow their frequent practice of showing the output at the intermediate tail-recursive stage before the final conversion to an iterative program in some ALGOL-style language. This makes the process and comparisons clearer.

For example, take a template <Input, Output, Constraints> that corresponds to the insertion and use of an accumulator, such as $< \Sigma_1, \Sigma_1', \Xi_1 >$ from Huet & Lang:

Σ_1: $f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x)))$

Σ_1': $f'(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ g'(e(x), d(x))$

$g'(x,y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g'(e(x), h'(y, d(x)))$

Ξ_1: $\forall x \forall y \forall z\ h(x, h(y,z)) = h(h'(x,y), z)$

$\forall x\ h(x, \perp) = \perp$

where $\perp$ is the base element of the domain of h's second argument.

Notice that their problem is already significantly different from Darlington's because of the existence of the new variable h' in the output schema which will not be identified by any matching between the input schema and the input. This

is a direct consequence of generalising templates. They become more expressive, but this brings a concomitant larger search space.

The result of second-order matching between the input schema and the definition of *reverse* (8.10) given earlier:

$$reverse(l) \Leftarrow \underline{if}\ null(l)\ \underline{then}\ nil\ \underline{else}\ reverse(cdr(l)) <> cons(car(l), nil)$$

(they assume that f is *reverse* and x is l) is a number of partial substitutions. These are identical in some of their values:

x	a	b	e	f
l	$\lambda u.null(u)$	$\lambda u.nil$	$\lambda u.cdr(u)$	$\lambda u.reverse(u)$

but each partial substitution offers differing values for d and h:

	d	h
1.	$\lambda u.cons(car(u), nil)$	$\lambda u \lambda v.v <> u$
2.	$\lambda u.car(u)$	$\lambda u \lambda v.v <> cons(u, nil)$
3.	$\lambda u.u$	$\lambda u \lambda v.v <> cons(car(u), nil)$

With these substitutions, the first of the associated conditions above becomes:

	$h(u, h(v, w)) = h(h'(u, v), w)$
1.	$(w <> v) <> u = w <> h'(u, v)$
2.	$(w <> cons(v, nil)) <> cons(u, nil) = w <> cons(h'(u, v), nil)$
3.	$(w <> cons(car(v), nil)) <> cons(car(u), nil)$ $= w <> cons(car(h'(u, v)), nil)$

The conditions must be shown to hold, and an identification of h' is needed. This is a MOR problem. They suggest that one way of doing this is to use the now partially instantiated constraints as match equations. Doing this means

that h' may only be identified as composed from functions appearing in the constraints. They cannot identify it as being composed from any other functions in the theory. Realistically, though, any solution available via this route is probably simpler than all the alternatives.

Unfortunately, this technique will only work for some of the templates. With the constraints above, it is very hard to see how matching alone could work, without using information about other theorems. In their paper they demonstrate how it might work on a template with more amenable constraints such as those from $< \Sigma_2, \Sigma_2', \Xi_2 >$

If we had chosen that template, we would have been able to use one of its constraints (see figure 8.3):

$$\lambda u.nil <> cons(car(u), nil) = \lambda u.h'(nil, u)$$

Matching offers four solutions for h':

$$\lambda y \lambda z.\ \ y <> cons(car(z), y)$$
$$\lambda y \lambda z.\ \ nil <> cons(car(z), y)$$
$$\lambda y \lambda z.\ \ y <> cons(car(z), nil)$$
$$\lambda y \lambda z.\ \ nil <> cons(car(z), nil)$$

The second of these leads to a viable solution. With it, the second condition becomes

$(nil <> cons(car(w), v)) <> cons(car(u), nil)$
$= nil <> cons(car(w), v <> cons(car(u), nil))$

Symbolic evaluation using the definition of $<>$ makes this:

$cons(car(w), v) <> cons(car(u), nil)$
$= cons(car(w), v <> cons(car(u), nil))$

which further evaluates to be true:

$cons(car(w), v <> cons(car(u), nil))$
$= cons(car(w), v <> cons(car(u), nil))$

The third condition is immediately true since <> is strict in its second argument.

Even with this template, there are other constraints to choose from to identify h', and one of them has h' on both sides of the equality, so that second-order unification, not second-order matching, would be required. In practice, each of their templates has a constraint where the unknown function features only once, but there is no guarantee that this will always be so.

In general, more than one of the various substitutions might lead to a successful solution. This is not of itself a problem, although selecting a "best" one might be. Detecting which substitutions do not yield successful solutions is more problematic. In cases described in their paper, more elaborate theorem proving was required, such as proving the associativity of <>. It is not clear how this would be handled: by recording standard results, trying out a few values, or a little theorem proving.

8.3.1 Search

Search arises at a number of points:

- Which template to attempt? Although the initial matching to input schema may be straightforward, all the variables must be identified and the constraints shown to be satisfied before a template choice can be deemed successful.

- Choice of initial match to follow up.

- Choice of constraint to use to suggest matches for the rest of the variables.

- Choice amongst solutions proposed by the matcher for the variables which don't appear in the input schema.

As can be seen from the example above, the search space is not immense. Taking the above example as a typical case suggests a size of the order of:

Number of templates (*6*) *

Number of initial matches (*3*) *

Number of constraints (*2*) *

Number of matches from constraints (*4*)

(= *144*)

8.4 Comparison of Template Systems with *MOR*

Each of Huet & Lang's static templates captures a tail-recursion transformation with some generality. Although, they discuss some of the problems involved in using the templates, they do not significantly address automation. Their main efforts are devoted to proving that template use preserves the equivalence of the input and output programs.

Using templates requires second-order matching to select an input schema. Matching an input schema only solves part of the problem. Except for the specialised Fibonacci one, all the templates have function variables in their output schema which do not appear in the input schema, and so will not be identified by any initial match (I am not counting the name of the new function being defined). These variables must be instantiated for the output schema to define a function. Huet & Lang suggest instantiating the variables by using the constraints on the functions appearing in the schemas. These constraints correspond to assumptions which were made in order to prove that the template transformation preserved the equivalence of input programs and output programs. The constraints describe general properties similar to associativity or functions being inverse of each other.

The process of identifying one of these variables works by finding a constraint where the variable only appears once, and assuming that the two sides of the equality (all constraints are equalities) can be matched. A substitution arising from the match is taken to instantiate the variable. In all but one of their

templates, there is a constraint where the new variable function appears on only one side of the equality. However, as I have already shown in working through this process by hand, it is not simple.

As I have already suggested, some problems are apparent in this stage of the process. Firstly, there may not be a constraint where the variable to be instantiated only appears once. Secondly, the two sides of the constraint may be equal but not unifiable. Thirdly, the function required could exist within the theory but not appear in the constraints, though intuitively this is unlikely, since the whole point of the constraints is to link the input to the output. Fourthly there may be different constraint equations to choose from, each of which may supply different substitutions, and it is not always obvious how we would choose between them.

Darlington's system also has constraint equations, but avoided this problem by storing (or asking for) standard general properties about functions, such as associativity or inverses, and checking against these known values. This seems a more sensible way of tackling the problem.

Apart from the ones about base values, all the constraint equations correspond to wave rules and base cases of function definitions.

So as a search problem, I believe that Huet & Lang have set themselves a harder task in general, in terms of identifying the new function. If it were attempted as Darlington does or as CLAM would, by searching a knowledge base for a suitable candidate, the various systems would have equivalent tasks.

The six templates cover the following cases:

1 & 2. The kinds of tail-recursion optimisation I have implemented using *MOR* with the CLAM system.

3. The conversion available when an inverse is known, so that calculation works from the base value, and works upwards.

4. A transformation designed to suit the fibonacci function.

5 & 6. These are similar to the first two, but make use of existing function arguments as accumulators, instead of introducing new ones.

Effectively the static templates preprocess what the MOR/CLAM system does dynamically for a number of fairly general cases. In principle, therefore, MOR/CLAM has greater potential, as it should be able to construct any of these, and any others which do not happen to fit one of these prepared templates. In practice, its ability to achieve this potential is restricted by:

- The form of the new sequent which is proposed, the proof of which is expected to identify the tail-recursive function.

  The current MOR/CLAM system only adds one new argument and expects it to function as an accumulator. In general one might have to add others fulfilling such rôles as initial values, or values when looping must stop. For the fibonacci style improvement extra storage arguments are needed, which achieve the same effect as tupling. Other functions can be optimised without any further arguments, only needing judicious use of the arguments they already have, as in templates 5 and 6.

- The guidance MOR receives to lead it to an overall proof which satisfies the requirements for tail-recursion may inhibit it from finding some solutions. This is true at present. Currently, for example, it insists that there be a transverse movement of the wavefront. This could theoretically be achieved instead by a longitudinal movement up, and then another one down.

Template systems are tied to the transformations they can represent through second-order patterns. Information about why these effect tail-recursion is fixed in the structure of the output schemas. Any automated systems based on templates would do no reasoning about how a transformation into a tail-recursive function should be composed, beyond instantiating variables with suitable values. Although the templates may span many common cases, they will always be limited to the forms of recursion schemes inherent in their description, and

to functions satisfying pre-selected constraints which permit a previously proved equivalence proof.

In proofs-as-programs and fold-unfold there is potentially far more flexibility, as it becomes possible to have automated systems which can adapt themselves to more complex recursion schemes, in the pursuit of the end result of a tail-recursively defined function equivalent to the original. Information about what constitutes the overall goal can be explicitly available for reasoning.

8.5 Ensuring Equivalence of Programs

Ensuring equivalence of programs using schemas is one of Huet & Lang's central concerns. They prove that their input and output schemas are equivalent under any semantics which model the constraints and the language. Their overall operation is split into stages, first achieving tail-recursion and then relying on standard transformations to convert this to an iterative version. The equivalence proof covers the naïve to tail-recursive part of the translation.

To ensure the validity of each template $< \Sigma, \Sigma', \Xi >$ they expect to show that

$$\mathcal{M}, \Xi \models \Sigma = \Sigma'$$

where $\mathcal{M}$ is the set of interpretations which satisfy the axioms of the language as described in whatever logical system is being used, e.g. LCF. $\mathcal{M}$ should be closed and not refer to the free variables of the schemas which will be instantiated using some substitution σ, which will apply them to any particular program fragment. Since the constraints in $\sigma\Xi$ must be valid, they are some of the theorems. Indeed if the constraints weren't provable, the schemas wouldn't be equivalent either, as the constraints are a by-product of schema equivalence validation. So for any valid template, $\mathcal{M} \models \sigma\Xi$. This information is crucial in guiding the $\mathcal{MOR}$ process too, because it means that we can expect to find the information about our mystery meta-variables from the associated theory, just as the template-users do.

Huet & Lang show that

$$\mathcal{M}, \Xi \models \Sigma = \Sigma'$$

for each of their templates. To illustrate this, I shall now work through the example proof they describe, as it will be instructive to compare its structure with the ones MOR uses.

Take the template:

Σ_1: $f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x)))$

Σ_1': $f'(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ g(e(x), d(x))$ where

$g(x, y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g(e(x), h(y, d(x)))$

Ξ_1: $\forall x \forall y \forall z. h(x, h(y, z)) = h(h(x, y), z)$ Ξ_{1a}

$\forall x. h(x, \perp) = \perp$ Ξ_{1b}

To show that

$$\Sigma = \Sigma'$$

they must show that

$$\forall x. f(x) = f'(x)$$

i.e.

$$\forall x. \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x))) = \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ g(e(x), d(x))$$

This is easy if we have:

$$\forall x \forall y. h(x, f(y)) = g(y, x) \tag{8.12}$$

The generalisation involved in this last step is more natural than it might appear, since e will normally correspond to a destructor function, and the argument positions in which it lies are intended to be on the primary recursion paths for the expressions they lie in. This step is necessary in order to ensure that the

proof of equivalence has covered all possible values. (8.12) is the key stage of their proof. They prove it by parallel computation induction. This is a very powerful induction scheme which leaves the variables universally quantified, and inducts over the development of the definition of the function. The essential structure of the computation induction proof is very similar to the fold-unfold process they use to check their templates. They use it because it is clearer and less long-winded than the full induction proof. Examining one of these fold-unfold checks will display the essential form and show some interesting parallels.

Starting with

$$\underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x)))$$

they take, as they say, a eureka step (handy for 8.12), and **define**

$$g(x, y) \Leftarrow h(y, f(x))$$

They do not explain how they pick just this generalisation or why they design it to include y, but there would be two problems if they chose to prove just $\forall x.h(d(x), f(e(x))) = g(e(x), d(x))$:

- the induction would not be guaranteed to cover all cases of h or g, it would cover cases of d or e. so their formal correctness would be spoilt;

- it would not be possible to use the induction hypothesis in the step case of the computation induction proof, since the more constrained instances would not permit matching at the appropriate stage.

There is an obvious similarity between this step, (8.12) and the key stage of the corresponding $\mathcal{MOR}$ proof, where the original goal $\forall y \exists z.z = f(y)$ is replaced by a new goal of proving $\forall y \forall a \exists z'.z' = F(f(y), a)$. In cases such as the *reverse* optimisation, F corresponds to h, and to build the constructive witness for z' we construct g and use it to supply z'. The motivation for introducing a as a new variable was exactly in order to have a more powerful inductive structure, though it is quite a different induction from the computation induction here.

The checking then takes this definition and by some folding, unfolding and use of lemmas produces a recursive equivalent with the desired property that the recursive definition's step case has no functions surrounding the reference to the recursive application of g:

unfolding (acts like an induction, I shall insert the wavefront):

$g(x,y) = h(y, f(x))$

$= h(y,\ \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ \boxed{h(d(x), \underline{f(e(x))})})$

"rippling" the if construct outwards:

$= \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ h(y, \boxed{h(d(x), \underline{f(e(x))})})$

using Ξ_{1a} (rippling sideways):

$= \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ h(\boxed{h(\underline{y}, d(x))}, f(e(x)))$

folding with the definition of g (the equivalent of using the induction hypothesis) builds the accumulated part onto y:

$= \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g(e(x), \boxed{h(\underline{y}, d(x))})$

The parallels with a rippling-structured proof are very close, and the purpose of the steps is much the same. What we would term a wavefront is moved from being around the $f(e(x))$ term over to the new variable, so that it can be absorbed.

Huet & Lang are describing two processes, a formal equivalence proof, and a less formal fold-unfold checking, which they note "might be developed to help in discovering new templates". In effect, that is similar to what the MOR proof does, except that MOR skips the explicit template since it is working directly with the function to be optimised. Remember from chapter 7 that there are two proof branches, one that identifies a tail-recursive function, and another that justifies its equivalence to the original specification. MOR builds its information up through the identification branch, using other information from the theory, and the justification of equivalence is a secondary proof. Huet & Lang's fold-unfold checking is very close to Darlington's work on folding and unfolding proofs, and corresponds to the identification branch of one of my proofs. It is interesting to note that Huet & Lang use this to check their schemas because it's easier.

They then achieve the justification of equivalence of schemas by computation induction, the formal proof of which is complex. Of course they only have to prove schema equivalence once for each template.

Although these key stages have such non-coincidental similarity, this is an illusion in one respect. The operations are carried out at different levels.

Huet & Lang are operating in advance to prove that second-order templates have equivalent input and output schemas. They perform proofs on these schemas, and then apply the schemas to given first-order functions.

From the point of view of automation, each of these steps is hard. Their main proof effort takes place at what is effectively their meta-level. It involves "eureka" steps to find suitable intermediate lemmas, and neither finding these nor proving them is straightforward, as they say themselves. A significant amount of search and theorem proving would also be inherent to the application stage if it were automated. No attempt is made to automate the process of finding schemas.

All this is in some contrast with the proofs-as-programs style, where equivalence is a separate proof branch, operating on the fully instantiated object level formula. This justification subproof is automatically built in to the proof when the tail-recursive version is proposed. Provided that the new hypothesis describing the new tail-recursive function is used to justify the original conclusion (which gives the specification of the function to be optimised), equivalence is assured. I have already described such "justification" proofs, and beyond a little ingenuity to find suitable initial values, the proof involved is not too hard. Typically they involve instantiating universally quantified variables in the conclusion to match the hypothesis, initialisation, and some simplification.

The steps which Huet & Lang describe as "eureka" steps, have been automated in the *MOR* system.

8.6 Achieving Specification, Optimisation and Synthesis

In order to be able to separate the components out clearly and discuss their function, I will briefly re-iterate an account of the underlying logical operation of my system.

If we assume a function defined[2]:

$$f(x,\vec{y}) \Leftarrow \underline{if}\ \ b(x)\ \ \underline{then}\ \ h(x,\vec{y})$$
$$\underline{else}\ \ f'(f(d(x),\vec{y}),x,\vec{y})$$

where f' is recursively defined on its first argument, then f is non-tail-recursive.

The proof statement that this specification is well-defined, i.e. that it defines a total function whose output is well-formed, is:

$$\forall x \forall \vec{y} \exists z.z = f(x,\vec{y})$$

which is proved by supplying a function which constructs the z given x and $\vec{y}$. Clearly, many function descriptions are available which would generate a suitable z. The recursion equations hold for these functions too. There is a trivial proof by simply using f itself, and taking z to be $f(x,\vec{y})$, but that does not give a new function.

We need to introduce such a new function into the proof, and do so in such a way that it is synthesised, not just verified to be the equivalent of f. If a new function, f_t is introduced as:

$$\forall x \forall \vec{y}.f_t(x,\vec{y}) = f(x,\vec{y}) \tag{8.13}$$

---

[2]I am ignoring a more general recursive description for the sake of clarity, it does not affect the argument

then the proof will verify the equivalence of the two functions, but without ensuring a specification and synthesis of f_t. This can be achieved by using the cut rule to insert a new specification proof node:

$$\forall x \forall \vec{y} \exists z.z = f_t(x, \vec{y})$$

and then this becomes an extra hypothesis in the proof of (8.13). An accompanying proof branch will retain the original proof goal, and include this as a new hypothesis.

The verification part of the proof takes place in the second branch, where the new hypothesis is made to generate its values; they are used as evidence for the conclusion, and verified to be the same.

To help with proof guidance, the synthesis branch can be more suggestively specified, as it is for the middle-out proofs. These suggestions incorporate some of the guidance material which in other systems would be in schemas, such as the existence of accumulators. Other guidance is built into the structure of the methods, as has been described in an earlier chapter, and implicitly into the extent of the theory loaded up and made known. A copious body of theorems will not lead the system into endless search, but an inadequate one will fail to feed it the information it needs to proceed. It would be a significant extension to enable it to speculate and prove such lemmas as it needed on the fly.

It is important to note that the guidance from these suggestions need not be restrictive, as schemas are. It need not correspond to any particular schema at all except insofar as it introduces and dictates the use of new variables. The inserted statement from which the synthesis proceeds is of the form:

$$\forall x \forall \vec{y} \forall a \exists z.z = F(f(x, \vec{y}), a)$$

This "suggestion" actually corresponds to the first stage of Huet & Lang's general equivalence proof which they must undertake separately for each schema. The form of this varies necessarily from schema to schema. The rippling structure echoes the structure of their proofs.

Consequently, the middle-out approach encompasses a number of schemas by having a pattern for constructing an equivalence justification, and only looking for such constraints as it needs for this actual proof from the supplied theory. Indeed it should be more flexible as it will not be tied to any particular schema and its validation constraints. One could either regard this as skipping the schema stage altogether, or alternatively as dynamically constructing the essence of a schema.

Using templates, the result achieves tail-recursiveness because this is an observed property of its output schema. Each template is constructed as a generalisation of just such observations. The MOR/CLAM proofs-as-programs implementation binds the definition of tail-recursion in and means that tail-recursiveness is an assured property if the method succeeds.

8.7 Static and Dynamic

It is now possible to see the proofs-as-programs work possibly as custom-building templates dynamically, or more accurately as avoiding the need for templates at all, as Darlington's fold-unfold work does. This is as opposed to the use of pre-defined static schemas to describe the kinds of conversions which will effect optimisation, the approach taken by Darlington's thesis work, Huet & Lang and others.

By defining templates in advance, they are obliged to predict the exact course of the equivalence proof between the original expression and the converted form for anything using this template. This proof will require certain properties of the new functions, such as their associativity, and these are noted as a list of constraints to be satisfied. It is in the other proof, the identification proof, that the real work is done. That constructs the rippling pattern of the function that gets constructed.

Generality of schemas is a mixed blessing, as I have shown. Those templates which are most powerful are also the hardest to use because they involve the

most variable quantities to be established. This is a trade-off which must be gauged for any automated system.

The dynamic systems, using MOR or fold-unfold, can adapt to a variety of recursion schemes, and make use of such properties as are true of the object level formulae they are working with. They are not constrained by a need to complete general purpose second-order proofs.

8.8 Range of Problems Covered

I have comparable examples for the first two of Huet and Lang's templates, but not for the others, which, apart from the third do not use accumulators. They give no examples for most of their templates.

I shall give any relevant definitions and lemmas with each example, converted to destructor-style description, to ease comparisons, although the definitions I use are constructor style. Appendix B shows some example synthesis plans, and the definitions and lemmas typically available.

To summarise common definitions and lemmas - in all the rest of this section, p and s are the predecessor and successor functions respectively on the natural numbers, *plus*, *times*, *zero* and <> (append) are defined as

$$
\begin{aligned}
plus(x,y) &\Leftarrow \underline{if}\ zero(x)\ \underline{then}\ y\ \underline{else}\ s(plus(p(x),y)) \\
times(x,y) &\Leftarrow \underline{if}\ zero(x)\ \underline{then}\ y\ \underline{else}\ plus(times(p(x),y),y) \\
zero(x) &\Leftarrow \underline{if}\ x = 0\ \underline{then}\ pnat\ \underline{else}\ void \\
(x <> y) &\Leftarrow \underline{if}\ null(x)\ \underline{then}\ nil\ \underline{else}\ cons(car(x), cdr(x) <> y)
\end{aligned}
$$

The associativity of *times*, *plus* and <> are known. *nil* is the empty list, *null* the function that tests for the empty list, and *car*, *cons* and *cdr* the usual list functions.

8.8.1 Template 1

Σ_1:	$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x)))$
Σ_1':	$f'(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ g'(e(x), d(x))$
	$g'(x,y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g'(e(x), h'(y, d(x)))$
Ξ_1:	$\forall x \forall y \forall z\ h(x, h(y,z)) = h(h'(x,y), z)$
	$\forall x\ h(x, \perp) = \perp$

Example:

$$reverse(l) \Leftarrow \underline{if}\ null(l)\ \underline{then}\ nil\ \underline{else}\ reverse(cdr(l)) <> cons(car(l), nil)$$

$$reverse_{tr}(l,a) \Leftarrow \underline{if}\ null(l)\ \underline{then}\ a\ \underline{else}\ reverse_{tr}(cdr(l), cons(car(l), a))$$

8.8.2 Template 2

Σ_2:	$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b\ \underline{else}\ h(x, f(e(x)))$
Σ_2':	$f'(x) \Leftarrow g'(x, b)$
	$g'(x,y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g'(e(x), h'(y, d(x)))$
Ξ_2:	$\forall x\ h(x,b) = h'(b,x)$
	$\forall x \forall y \forall z\ h(x, h(y,z)) = h(h'(x,y), z)$
	$\forall x\ h(x, \perp) = \perp$

Example:

$$factorial(x) \Leftarrow \underline{if}\ zero(x)\ \underline{then}\ s(0)\ \underline{else}\ times(factorial(p(x)), x)$$

$$factorial_{tr}(x,a) \Leftarrow \underline{if}\ zero(x)\ \underline{then}\ a\ \underline{else}\ factorial_{tr}(p(x), times(x,a))$$

8.9 Higher-Order Problems

The MOR system can optimise the (third-order) summation functional

$$\sum_{i=0}^{n} f(i) \quad \Leftarrow \quad \underline{if}\ zero(n)\ \underline{then}\ f(0)\ \underline{else}\ plus(f(n), \sum_{i=0}^{p(n)} f(i))$$

Although this is very close to template 1 in form, and requires no more than second-order matching, it is parameterised over a variable function, f, which does not become instantiated in order to optimise the definition of the functional, Σ.

In general, operating in a higher-order context should not affect the MOR system.

8.10 Using a Proof Planning and Development System

By operating within a proof-planning system it was possible to experiment with these ideas fairly easily by using the method definitions. A more flexible planning structure would have been useful, allowing heuristics to decide whether the result of applying a method had been informative in the sense of instantiating variables reasonably certainly, with a small search space. If that were not the case, such a flexible planner would have postponed this proof branch while another was attempted which cause less instantiated branching of the search space. This notion of informative and uninformative inference attempts was pertinent in the case of the base and step case choices, where the step case was artificially supplied to the planner first, because it could supply the information to instantiate the meta-variable correctly. Using the base case would have tried every function in the library which could match because it had so few real constraints in it.

Using CLAM meant that a sizeable body of theory could be made available. Without the definition of purpose built into the proof plan, however, that volume of information would be impossible to negotiate automatically.

The meta-variables are available for explicit reasoning. Proofs convert directly into executable programs, via the Curry-Howard isomorphism.

8.11 Use of Unification

Unification is far too prolific in its solutions to be used easily without extra control. Darlington, Huet & Lang and I all reduce the problems to matching wherever possible because of this, using F-matching, second-order matching and ω-order matching respectively. Apart from its inefficiency, there is no disadvantage in using a higher-order unifier, as it only produces the same solutions that would have been proposed by the unifier appropriate for the level of problem supplied. I.e. if the highest order of variable is only second-order, it operates as a second-order algorithm.

Further, we all reject solutions which are valid as unifiers, but not valid within the context of our problems, expressly those which build parameter variables into the definition of some variable function. As I have noted earlier, this is Dale Miller's temporal scoping. This restriction reduces the number of solutions considerably.

The move from F-matching to full matching[3] enables matches where variables must be instantiated to a composite function, as is required for some problems, for example an additive version of multiply a number (y) by the integer half of another (x):

$$\begin{aligned} multhalf(0,y) &\Leftarrow 0 \\ multhalf(s(0),y) &\Leftarrow 0 \\ multhalf(s(s(x)),y) &\Leftarrow plus(multhalf(x,y),y) \end{aligned}$$

for which $\mathcal{MOR}$ can generate a tail-recursive function:

$$multhalf_{tr}(0,y,a) \Leftarrow a$$

---

[3]for which algorithms only became available after Darlington's thesis

$$multhalf_{tr}(s(0), y, a) \Leftarrow a$$
$$multhalf_{tr}(s(s(x)), y, a) \Leftarrow multhalf_{tr}(x, y, plus(y, a))$$

8.12 Conclusion

What, then, are the responses to the key questions posited at the beginning of this chapter, in the light of this new work? How do the various approaches relate to each other?

Proofs-as-programs makes proof of equivalence of programs to each other or to their specification an inherent feature, for example an extracted program is guaranteed to satisfy its specification. When we introduce a new formula to describe a tail-recursive algorithm, its equivalence to the original specification is a subgoal. Such subgoals are generated automatically, and are usually fairly straightforward to prove. Other systems have to prove the equivalence preserving properties of the transformations. When this ease of equivalence is taken into account as well as the ability to characterise and detect tail-recursion, we can see that any completed synthesis proof can easily be checked for the tail-recursive qualities of the resulting *proof*. So, for example, earlier work of mine on synthesis of programs involving case-splits (not involving MOR and not reported in this thesis) such as greatest common divisor, would readily be checkable for tail-recursiveness.

The proofs-as-programs approach makes it possible to describe the proof forms corresponding to tail-recursion very clearly. Work is still required to find such proofs, of course. A significant advantage of this approach can be seen to be that it gives us a general characterisation of our goal simply by restricting our attention to a class of object level proofs. Other approaches have to attempt this through other means. The fold-unfold approach uses analogous procedures to ours (Burstall and Darlington), but they lack the information available through CLAM about progress and direction. In fold-unfold, the criterion for tail-recursiveness must be decided of the resulting defined function.

The template-based systems specify sets of transformations, which iff applicable achieve the desired result [Darlington 72, Huet & Lang 78, Cooper 66].

Although the analysis of the proof structure shows that it describes tail-recursion, the implementation and representation chosen have some built-in assumptions which restrict it from achieving the full power of the proof structure. Since it automatically builds in an accumulator it will miss certain solutions where simply substituting functions would be adequate. Given the straightforwardness of characterisation of tail-recursiveness, it would not be difficult to search for such direct tail recursive proofs too. In these cases, however, a similar alternative with an accumulator can be found.

The fold-unfold approach has considerable versatility, in that it can work with whatever information is to hand, and can combine a number of program-optimising features. It lacks automated guidance at the level of our supermethods, and the ability to surmise the existence of, and then identify, entities needed for intermediate stages.

An important conclusion relates to the value of explicit templates. Although it might be suspected that their knowledge had just been slipped in somewhere else, that is not so. What *has* happened is that the meta-knowledge inherent in their validation proofs, which controls the identification of templates, has been embedded in the tail-recursive planning component. That now operates on actual theorems rather than on abstractions. The built in optimisation drives the process with guidance from the theory, and a customised validation is produced. The intermediate structure of the template becomes redundant. This would seem to be confirmed by the success of Darlington's later, non-template based, fold-unfold work.

My main conclusion is that a MOR/proofs-as-programs approach contains most of the best features of templates and fold-unfold. It has fold-unfold's flexibility over unforeseen recursive schemes, and the template approach's ability to supply values for unknown functions. MOR/proofs-as-programs has the added advantage that it can use an explicit characterisation of suitable proof struc-

tures for strategic guidance, aided by CLAM's tools for monitoring progress in such strategies.

My long list of points to be considered ended with a miscellany of larger issues which are important, but most of which are beyond the scope of what is still an experimental system.

- Completeness and Termination. These are not guaranteed, any more than for any other system addressing this kind of problem.

- Transparency of the Resulting Program. This is poor, but that is a well-known pitfall of the optimisation process. It is exactly the reason why optimisation is a good thing to automate, so that humans don't have to face this ugly side.

- Explainability of the Process. This is debatable, but by using more large scale "supermethods" can probably be improved.

- Choosing Between Alternative Solutions. I don't do this at all, I just stop with the first one. It could be an avenue for further work.

- Other Types of Optimisation. Templates can be used to achieve other kinds of optimisation. There is no reason why proof structures should not do this too. The problem of preferring one optimisation to another would, of course, still be present.

- Evaluation System. I have assumed lazy evaluation throughout. If some other form of evaluation were required, such as call-by-value, this type of approach should still work, but the proof structure would need some adaptation.

- Closeness to Any Actual Language or Implementation. The output of this system is a simple functional programming language, which can be executed, though it's not built for efficiency. It wouldn't be too hard to convert it into a more carefully implemented functional language.

- Efficiency of the Synthesising Operation. This is poor. Once it hits a lot of lemmas, it is dreadfully slow. However, it could be improved by some optimisation of the unification system, and streamlining the preconditions in the methods.

- Modifiability and Extensibility. Both the flexibility of arguing about proof structures, and the powerful tool of the proof planner make this approach highly modifiable and extensible.

As I hinted at the beginning of this chapter, different purposes emphasise different aspects of work. It should now be clear that my work is strongly directed at automation and discovery, ensuring equivalence is a necessary sideline, and even easier when dealing with actual functions rather than templates. The concentration on formal correctness of templates in Huet & Lang's work, becomes redundant in an approach which avoids such intermediate representations. Proof planning has given us the tools to characterise the structure of the proofs we need, and hence their programs. Discovery arises out of the interaction of a theory with the demands of optimisation. Its automatic organisation can be controlled effectively by the $\mathcal{MOR}$ process.

Σ_1:	$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ h(d(x), f(e(x)))$
Σ_1':	$f'(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x)\ \underline{else}\ g'(e(x), d(x))$
	$g'(x,y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g'(e(x), h'(y, d(x)))$
Ξ_1:	$\forall x \forall y \forall z\ h(x, h(y,z)) = h(h'(x,y), z)$
	$\forall x\ h(x, \perp) = \perp$
Σ_2:	$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b\ \underline{else}\ h(x, f(e(x)))$
Σ_2':	$f'(x) \Leftarrow g'(x, b)$
	$g'(x,y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ h(y, b(x))\ \underline{else}\ g'(e(x), h'(y, d(x)))$
Ξ_2:	$\forall x\ h(x,b) = h'(b,x)$
	$\forall x \forall y \forall z\ h(x, h(y,z)) = h(h'(x,y), z)$
	$\forall x\ h(x, \perp) = \perp$
Σ_3:	$f(x) \Leftarrow \underline{if}\ x = a\ \underline{then}\ b\ \underline{else}\ h(x, f(e(x)))$
Σ_3':	$f'(x) \Leftarrow g'(a, b, x)$
	$g'(x,y,z) \Leftarrow \underline{if}\ x = z\ \underline{then}\ y\ \underline{else}\ g'(e'(x), h(e'(x), y), z)$
Ξ_3:	$\forall x\ e'(e(x)) = x$
	$\forall x\ e(e'(x)) = x$
	$\forall x\ h(x, \perp) = \perp$

$\perp$ is always the base element of the relevant domain

Figure 8–1: Huet and Lang's Templates 1-3

Σ_4:	$f(x) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b\ \underline{else}\ h(f(d(d(x))), f(d(x)))$ (fibonacci)
Σ'_4:	$f'(x) \Leftarrow g'(x, b, b)$
	$g'(x, y, z) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ y\ \underline{else}\ g'(d(x), h(z, y), y)$
Ξ_4:	$\forall x\ (\neg a(x) \wedge a(d(x))) \supset a(d(d(x)))$
	$\forall x\ h(x, \bot) = \bot$
Σ_5:	$f(x, y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x, y)\ \underline{else}\ h(y, f(c(x), d(y)))$
Σ'_5:	$f'(x, y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x, y)\ \underline{else}\ f'(c(x), h'(y, d(x)))$
Ξ_5:	$\forall x \forall y \forall z\ h(x, h(y, z)) = h(h'(x, y), z)$
	$\forall x \forall y \forall z\ h(x, b(y, z)) = b(y, h'(x, z))$
	$\forall x\ h(x, \bot) = \bot$
Σ_6:	$f(x, y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x, y)\ \underline{else}\ h(x, f(e(x), y))$
Σ'_6:	$f'(x, y) \Leftarrow \underline{if}\ a(x)\ \underline{then}\ b(x, y)\ \underline{else}\ f'(e(x), h'(x, y))$
Ξ_6:	$\forall x \forall y \forall z\ h(x, h(y, z)) = h(h(x, y), z) \vee \forall x \forall y \forall z\ h'(x, h'(y, z)) = h'(h'(x, y), z)$
	$\forall x \forall y \forall z\ h(x, b(y, z)) = b(y, h'(x, z))$
	$\forall x\ h(x, \bot) = \bot$

$\bot$ is always the base element of the relevant domain

Figure 8–2: Huet and Lang's Templates 4-6

Chapter 9

Middle Out Reasoning to Guide Generalisation for Induction

I shall use the term generalisation in its mathematical sense, not in any broader sense used in Artificial Intelligence.

The attractions of generalisation are that when applied correctly, it yields a more powerful result, and perhaps an easier proof. Indeed many theorems are not provable at all without generalisation.

The difficulties of generalising and automating generalisation are in deciding the following questions

- when is it appropriate to generalise?
- what expressions should be generalised?
- what should they be generalised to? It is very easy to create a new formula, of which the original is an instance, but which is no longer a theorem.
- how and when can we tell some generalisation is succeeding?

To answer these questions, and control proofs, it is invaluable to rely on a general structure for a type of problem. My attempts have taken advantage of choosing a particular class of problems (universally quantified formulae requiring induction) and developing an understanding of properties and structures of their

proofs. As a result, there are other classes of generalisation, outlined in sections 9.1 and section 9.2, that my techniques do not address, such as removal of redundant hypotheses, or some kinds of proofs involving existentials.

Previous attempts to automate such generalisation have been designed for specialised classes of generalisation problem. As described in chapter 2, for instance, they look for multiply occurring terms, and then use heuristics to decide *whether* to generalise, and if so, *how* to generalise the theorem. Although generalisation is used as an aid to induction, and the heuristics are built with an eye to future inductions, this relationship to induction is not fully exploited. The heuristics work in advance of the subsequent induction proof, not alongside it.

Using MOR, we can bind the generalisation choices more closely to the needs of induction. We can use the meta-level to describe a generalisation flexibly, but postpone making a precise choice. Subsequent proof planning selects the particular generalisation necessary for success.

I will try to answer the questions I posed at the beginning of this chapter. To do that, I will first give a brief account of what constitutes generalisation. Section 9.2 will briefly review how generalisations are effected within a sequent calculus, and assign some limits I will assume for my task. Following that, section 9.3 will describe the form my flexible description will take, and why I believe it is adequate for its purpose. Section 9.4 describes how we can decide that an induction is working or being ineffectual. This helps us know when to attempt a generalisation and assess whether one is proving worthwhile. Section 9.5 analyses the generalisation process under MOR in detail, showing what constraints are available and discussing search control. Subsequent sections demonstrate the technique working on a variety of theorems.

Comparison between MOR and other approaches will be addressed in chapter 10.

9.1 What is Generalisation?

Generalisation is a proof step which allows us to postulate a new theorem as a substitute for the one we are currently trying to prove, and then use it to justify the original. Usually, the generalisation is inserted into the proof giving the theorem prover two tasks, to prove the generalisation and show that it justifies the original theorem. In a sequent calculus this insertion is commonly effected by using the cut rule. The cut rule allows us to add a new hypothesis to the sequent being proved, provided that we can give a (separate) proof of the new hypothesis from the existing hypotheses. An exception might be if the generalisation corresponds to a primitive rule of inference. Use of the cut rule is unnecessary, for example when proving an implication by proving its consequent without reference to the antecedent. Such examples can reasonably be neglected here.

The existence of a cut elimination theorem for a formalised theory is an indicator of whether or not generalisation is necessary. Cut elimination theorems are proof theoretic results, proved inductively over the formulae of the logical formalism and the axioms of a theory. When they exist, they show that any theorem of the theory can be proved without resort to the cut rule of inference. If it can be shown that the cut rule is not required, then neither is generalisation. Cut elimination theorems are not available for most theories.

There are two main reasons why generalisation is such a valuable technique in mathematics:

- It may take the insights from a specific proof and create a more powerful and expressive result. This may involve anything from dropping an unnecessary condition to grand analogies yielding a result valid for all the entities of some type instead of just a particular one.

- Often, within the rigours of mathematical logic, certain theorems are only provable via a generalisation. Only theories for which a cut-elimination

Original	Generalisation		
P	$P \wedge R$		9.1
$R \rightarrow P$	P		9.2
$r > t$	$r > s \wedge s > t$		9.3
$\forall x.Q(x,x)$	$\forall x \forall a.Q(x,a)$		9.4
$\forall x \forall z.Q(x,f(z))$	$\forall x \forall a.Q(x,a)$		9.5
$\forall x.Q(x,x)$	$\forall x \forall a.Q(x,g(x,a))$	(if $\exists b.g(x,b) = x$)	9.6
$\forall x.Q(x,c)$	$\forall x \forall a.Q(x,a)$		9.7
$\exists x \exists y.Q(x,y)$	$\exists x.Q(x,x)$		9.8
$\exists x \exists y.Q(x,y)$	$Q(0,c)$		9.9

where P and R are propositions, Q is a 2 argument predicate, g is a function, and r, s and t are free variables, and c is a constant.

Figure 9–1: Examples of Generalisations

theorem is available can be guaranteed not to need operations we would normally describe as generalisations.

Generalisation is widely used as a concept, but exactly what it means is not tightly defined. If the new theorem postulated bore little relation to the original theorem to be proved, for example if it were some ancillary lemma, we would not count it a generalisation. So we do not immediately class as a generalisation any theorem whose addition to the current hypotheses makes the current goal provable. In some way, we expect the new theorem to imply the original, and to achieve that implication mainly "unassisted". This happens in some cases, such as those for which only instantiation is required. For others, an arbitrary amount of computation using definitions may be required to establish the justification, e.g. (9.6) in figure 9.1. This could be viewed as some kind of instantiation by unification using a built-in theory, but controlling such unification would be very hard. All the examples in Figure 9.1 might be taken as generalisations.

One of the dangers of generalisation is that of over-generalising - generalising to a new statement which is not a theorem. This is easily done when following syntactic generalisation rules, and hard to guard against. I shall address this issue at various points in the course of this chapter and the next one.

9.2 Logical Basis of Generalisation

Apart from those operations which might be regarded as null generalisations, because they are just the application of rules of inference, generalisations involve using the cut rule. Although we typically think of generalisation as taking a formula and somehow strengthening it, the examples in figure 9.1 show that this can take many forms. Sometimes distinguishing multiple occurrences generalises, sometimes it specialises, according to the polarity of the formulae concerned. All this become clear if we look at the nature of the justification and the cut rule.

If we consider this process in relation to the sequent calculus formulation, the only way of inserting formulae into the proof which are not subformulae of the current sequent, is the cut rule:

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash \Lambda}{\Gamma \vdash \Lambda}$$

Γ and Λ are arbitrary lists of formulae, and F is an arbitrary formula. For the purpose of generalisation, Λ will be a singleton, the formula which will be generalised, and F will be the formula which is its generalisation. Then the left-hand proof branch proves the generalisation, F, and the right-hand proof branch is a justification that F is indeed a generalisation, i.e. that its addition to the hypotheses results in a proof of the original sequent.

Starting from this point, we can look at generalisation as a use of the cut rule where F is very similar to some other formula in the sequent, in the sense I outlined earlier of something akin to unification with some theory built in. For

example Boyer and Moore's generalisation (9.5) from figure 9.1 is proved like this:

$$\frac{\Gamma \vdash \forall x \forall y.Q(x,y) \qquad \Gamma, \forall x \forall y.Q(x,y) \vdash \forall x \forall z.Q(x,f(z))}{\Gamma \vdash \forall x \forall z.Q(x,f(z))} \tag{9.1}$$

Aubin-style generalisation (9.4 in figure 9.1) follows the same pattern:

$$\frac{\Gamma \vdash \forall n \forall m.Q(n,m) \qquad \Gamma, \forall n \forall m.Q(n,m) \vdash \forall n.Q(n,n)}{\Gamma \vdash \forall n.Q(n,n)}$$

At the same time, some restrictions are apparent, regarding the polarity of the formula being generalised. The following inference steps are generalisations too:

$$\frac{\Gamma \vdash \forall n.Q(n,n) \rightarrow P \qquad \Gamma, \forall n.Q(n,n) \rightarrow P \vdash \forall m \forall n.Q(m,n) \rightarrow P}{\Gamma \vdash \forall m \forall n.Q(m,n) \rightarrow P}$$

$$\frac{\Gamma \vdash \exists n.Q(n,n) \qquad \Gamma, \exists n.Q(n,n) \vdash \exists m \exists n.Q(m,n)}{\Gamma \vdash \exists m \exists n.Q(m,n)}$$

In each case, the requirements of the justification branch insist that only some similar expressions are actually generalisations, those of which the originals are logical consequences.

For most induction proofs which I will consider, this point never arises, since we are dealing with a single expression on the right-hand side, which is universally quantified. Therefore the first (9.1) of these generalisation inferences covers them.

In general though, it would be necessary to consider the quantification and polarity of an expression carefully, in order to decide what was a generalisation, and would therefore result in a suitable justification branch.

First order predicate calculus with no theory axioms, is an example of when it is possible to prove that the availability of the cut rule does not add to the

body of theorems. Without it, all the same theorems can be proved. In such cases, generalisation in this sense is unnecessary from a purely logical point of view, though it may not be heuristically, or aesthetically.

9.3 Generalisation for Induction Proofs

Working in the context of plans for induction proofs, I am interested in the generalisations which facilitate such proofs. This perspective is helpful in guiding the generalisation process. My approach has been to find some way of casting problems in a flexible form so that the requirements of a successful induction could manifest themselves, and prescribe the generalisation required. I regard a successful induction as one which leads to the use of the induction hypothesis, and then (if the proof branch is not already finished) some *cancellation* - use of the functional or predicate substitution axioms, which eliminates some of the term structure present in the induction hypothesis.

Broadly, I expect to hypothesise suitable generalisations of types (9.4) to (9.6) of figure 9.1. Although they look different, the generalisations can all be viewed as cases of the following:

- Take terms in the expression to be generalised, and link them to a new universally quantified variable by meta-variable functions.
- Instantiate the meta-variable functions as required to allow the proof to go through. The instantiation may make the meta-variable functions projections onto some argument, or create a link between the term and the variable. The meta-variable will become this link.

So, for example, let us start with a formula $e[t,\ldots,t]$, with occurrences of a subterm t, which is a constant, free variable, universally quantified variable or non-atomic term containing no existentially quantified variables. Then for some variable a not occurring in $e[t,\ldots,t]$, $\forall a.e[F_1(t,a),\ldots,F_n(t,a)]$ is a flexible generalisation. It speculates about a possible generalisation. F_i are meta-variable

functions, indexed so that each occurrence of t can be treated differently. This assumes that a single new variable, a, will be enough to create a useful generalisation. I will return to this assumption later. The intention is that this construction inserts an object into the formula which can be used flexibly for meta-reasoning. Each F_i becomes identified as one of the following:

- $\underline{\lambda u \lambda v.u}$

  This means that $F_i(t, a) = t$, no generalisation.

- $\underline{\lambda u \lambda v.v}$

  Here, $F_i(t, a) = a$, generalisation of term t to the new variable, a.

- $\underline{\lambda u \lambda v.f(u, v)}$

  In this case, $F_i(t, a) = f(t, a)$, where f is some constant function symbol, or composition of such symbols. This permits the insertion of accumulators.

The higher order unification required to identify the F_i is described in chapter 4. Some further restrictions are appropriate to the unifiers which will be accepted. We are expecting unification to perform the above tasks *only*. We are not expecting it to suggest other alterations to the formula, such as introducing more occurrences of the unknown variables. Although such unifications would eventually be rejected since they would fail to lead to generalisations which justified the original formula, it is very expensive to consider them all. Consequently I am filtering out such unifications at present.

By creating this common approach, a variety of generalisations can be addressed, the exact form being directed by the needs of the proof. The new variable can be treated as an accumulator, a new induction variable, or simply as a variable on which no induction takes place, as required.

I will seek solutions which lead to a successful induction, rather than maximal generalisations. An advantage of this is that the number of new variables can be kept to a minimum. This also helps to ward off the ever present danger of over-generalisation.

9.4 When Induction Fails

If generalisation is to be viewed as a technique to fix failing inductions, we need a way of deciding that induction is failing. I shall devote this section to looking at what induction is and how it works.

9.4.1 How Does Induction Work?

The point of induction is to give us a strong hypothesis, similar to the theorem being proved. In constructor-style induction, we suffer for this by accepting a more complex theorem to prove. If as a result we fail to simplify the expression to be proved in any sense, by use of an induction hypothesis, then it is likely that our task has been complicated to no good effect and the induction attempt was faulty.

It's helpful to look at this first from the point of view of a theorem whose dominant functor (after quantification) does not readily admit a rewriting use within the theory, not $=$ or $<$ for example. This corresponds to permitting only strong fertilization in CLAM. The whole of the induction hypothesis must be used to substitute for (at least part of) the induction conclusion, after any necessary instantiation of quantified variables. This usage corresponds to functional and predicate substitution axioms.

Admitting rewriting allows the possibility that not all of the induction hypothesis need be re-created within the conclusion for the hypothesis to be used. In the case of $=$, manipulating the conclusion so that just the left-hand side of the hypothesis appeared as a subterm of the conclusion would allow that subterm to be rewritten. The right-hand side of the induction hypothesis equality would be substituted for the subterm. This is CLAM's weak fertilization. Its usage corresponds to transitivity axioms.

I will elaborate the basic possibilities ignoring the possibility of rewriting, and then go on to look at the effects of allowing rewriting afterwards.

9.4.2 Using Induction Hypotheses Which Cannot Be Used For Rewriting

Taking $\forall x \forall y P[x, y]$ as a theorem to be proved, a step case is of the form:

$$\forall y' P[x, y'] \vdash \forall y P[\boxed{c(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$$

where c stands for some construction inherent in the induction scheme, and y has been renamed as y' in the induction hypothesis to avoid ambiguity. The square bracket notation, as ever, indicates that P may be a composite function containing multiple occurrences of the arguments in the brackets. Although in principle there could be any number of other universally quantified variables present, i.e. instead of just y there could be $y_1, y_2, ...$, the essence of the explanation is clearer without them. The account extends straightforwardly, but with notational extravagance, to any finite number of arguments. The notation $\lfloor y \rfloor$ reminds us that y corresponds to a universally quantified variable (y') in the induction hypothesis; it is called a "sink" as any wavefronts which can be moved to immediately around y, can be "soaked up" by suitable instantiation of y'.

The only way the induction hypothesis can be used is if it can be made to unify with the induction conclusion, strong fertilisation in CLAM terms. Currently, the wavefront prevents this. There are two ways of rectifying this,

1. longitudinally rippling the conclusion out to $\vdash \forall y \boxed{c'(\underline{P[x, \lfloor y \rfloor]})}^{\uparrow}$ or

2. transversely rippling the conclusion sideways to $\vdash \forall y P[x, \left\lfloor \boxed{c''(\underline{y})}^{\downarrow} \right\rfloor]$. Then y can be introduced as a free variable, and y' chosen as $c''(y)$, provided all the term occurrences in the conclusion which match y' are the same. Then the hypothesis matches the conclusion.

In principle some combination of the two types of rippling may be required to achieve even one of these overall effects. For example both longitudinal and transverse rippling may be required to move a wave front sideways onto a sink. Indeed some combination of these two effects may occur, and one wave front

may move onto a sink, while another emerges from the original term. It can also happen that c' or c'' is null.

If the application of wave rules fails to achieve either of these two situations, the conclusion is strictly more complex than the one we started with, since it *is* the one we started with, with a wavefront inserted. So any subsequent proof attempt is likely to be at a disadvantage.

The only exception to this is if the effect of subsequent inductions is to combine with this failed attempt to create a complex induction scheme which was not part of our repertoire, but was needed at some rippling point in the expression. Perhaps an induction over more than one variable, or an n-step induction that had not been anticipated by the recursion analysis. By looking at the point at which rippling is blocked, we can often tell if that's the case, by seeing if there are any wave rules which could be enabled. Otherwise this induction has made the theorem more complex, and was therefore probably unwise. Abandoning it in favour of an alternative induction or generalising the theorem could enable a successful proof which uses the induction hypothesis.

9.4.3 Using Induction Hypotheses Which Can Be Used For Rewriting

With induction hypothesis rewriting available, the situation becomes more complicated, as more options, weak fertilisations, become available in addition to those in the non-rewriting case. Equality is the obvious example of this, and arises in most of our proofs. An induction step of the form

$$\forall y'.L[x,y'] = R[x,y'] \vdash \forall y.L[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor] = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$$

may only ripple far enough to enable the use of the induction hypothesis, (without loss of generality on the left), to permit a rewriting:

$$\forall y'.L[x,y'] = R[x,y'] \vdash \forall y.\boxed{k^1(\underline{L[x,\lfloor y \rfloor]})}^{\uparrow} = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$$

the rewriting makes the conclusion into

$$\vdash \forall y \boxed{k^1(\underline{R[x,\lfloor y \rfloor]})}^{\downarrow} = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$$

If we rewrite as justified by one side alone, does this bring us closer to our aim of simplifying the theorem to be proved?

I will return to the question of the use of the induction hypothesis in 9.4.6. First, I shall examine how rippling can act to inform that discussion. Particularly interesting are the instances in which cancellation becomes possible, i.e. reducing the proof of the equality of two terms with identical dominant functor, to proofs of the equality of corresponding arguments. This analysis is currently only partly implemented in CLAM.

There are six possible schematic combinations[1] of rippling progress on the two sides of the equality. Each side may become longitudinally or transversely rippled, or it may be blocked. By longitudinally rippled or transversely rippled I mean that one side of the equality in the hypothesis can completely match some (sub)term of the conclusion with suitable choices of universally quantified variables:

1. $\forall y \boxed{k^1(\underline{L[x, \lfloor y \rfloor]})}^{\uparrow} = \boxed{k^2(\underline{R[x, \lfloor y \rfloor]})}^{\uparrow}$

   Both sides are longitudinally rippled. Provided that the functor structure in k^1 and k^2 dominating $L[x, y]$ and $R[x, y]$ are (or can be made) the same, say we have:

   $$\forall y \boxed{k_f^1(\underline{L[x, \lfloor y \rfloor]}, \vec{k_a^1})}^{\uparrow} = \boxed{k_f^2(\underline{R[x, \lfloor y \rfloor]}, \vec{k_a^2})}^{\uparrow}$$

   where $k_f^1 = k_f^2$, then cancellation can be used, and one of the subgoals is just the induction hypothesis.

   The other subgoals ($\vec{k_a^1} = \vec{k_a^2}$) are the (hopefully) simpler task of showing the equivalence of the rest of the two wavefronts. If there are multiple occurrences of the induction and/or sinks present, this may not be trivial.

---

[1] It is of course possible for there to be multiple wavefronts and have some of them ripple longitudinally, and some transversely. I am not considering such complexities for now.

CLAM doesn't actually do this work to make wavefronts match. It leaves the matched terms in place and lets subsequent proof clear the discrepancies, quite possibly through further inductions, which can be confused by the continuing presence of the rewritten term. This is a common instance requiring Boyer and Moore style generalisation, and was part of the motivation for their technique.

It may be that this wavefront matching cannot be achieved just through cancellation, in which case the Boyer and Moore style generalisation would be essential.

2. $\forall y L[x, \left\lfloor \boxed{k^3(\underline{y})}^{\downarrow} \right\rfloor] = R[x, \left\lfloor \boxed{k^4(\underline{y})}^{\downarrow} \right\rfloor]$

Both sides are transversely rippled. If $k^3(y)$ and $k^4(y)$ are identical, then the induction hypothesis' universally quantified y variable can be instantiated to that value, and we have a hypothesis identical to the conclusion.

Otherwise, weak fertilisation, i.e. rewriting using the hypothesis (left-to-right, but the cases are symmetrical) makes this:

$\forall y R[x, \boxed{k^3(\lfloor \underline{y} \rfloor)}^{\downarrow}] = R[x, \left\lfloor \boxed{k^4(\underline{y})}^{\downarrow} \right\rfloor]$

and cancellation (functional substitution) simplifies this to showing the equivalence of the two wavefronts. The same comment applies as for the previous case.

3. $\forall y \boxed{k_1(\underline{L[x, \lfloor y \rfloor]})}^{\uparrow} = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$

One side is longitudinally rippled and the other is blocked. On rewriting the rippled side we get:

$\forall y \boxed{k_1(\underline{R[x, \lfloor y \rfloor]})}^{\uparrow} = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$

No cancellation is enabled immediately. But by rippling the wavefront back inwards on the left as far as possible, the outermost functor structure may be restored, and the cancellation enabled. At least some of the original term structure is removed, and the problem simplified.

4. $\forall y.L[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor] = R[x, \left\lfloor \boxed{k^4(\underline{y})}^{\downarrow} \right\rfloor]$

One side is transversely rippled and the other is blocked. On rewriting the rippled side we get:

$\forall y.L[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor] = L[x, \boxed{k^4(\underline{\lfloor y \rfloor})}^{\downarrow}]$

Depending on how much the blocked side has rippled, some substitution may or may not have been enabled. If not, this is exactly analogous to the previous case, and we would expect to try to reverse the movement of the wave front, and again ripple sideways, but backwards, to enable cancellation.

5. $\forall y \boxed{k^1(\underline{L[x, \lfloor y \rfloor]})}^{\uparrow} = R[x, \boxed{k^4(\underline{\lfloor y \rfloor})}^{\downarrow}]$

One side is longitudinally rippled and the other is transversely rippled, so we could rewrite either side:

- Rewriting the longitudinally rippled side gives:

$$\forall y \boxed{k^1(\underline{R[x, \lfloor y \rfloor]})}^{\downarrow} = R[x, \boxed{k^4(\underline{\lfloor y \rfloor})}^{\downarrow}]$$

- Rewriting the transversely rippled side gives:

$$\forall y \boxed{k^1(\underline{L[x, \lfloor y \rfloor]})}^{\uparrow} = L[x, \boxed{k^4(\underline{\lfloor y \rfloor})}^{\downarrow}]$$

Either way, cancellation is only going to be enabled by rippling k^1 back in and sideways towards the sink y, or by rippling k^4 up. If possible, these are probably worthwhile steps and may allow us to reduce the problem by cancellation. However, although some functions ripple both longitudinally and transversely, many do not. It seems likely that in the first case we have marooned a wavefront outside an essentially transverse system, and in the second, marooned one inside an essentially longitudinal system, around an argument position which was not part of the last induction, and so may not be in a good position to ripple away. It is unlikely that cancellation would be assisted in these circumstances, and they should be avoided.

6. $\forall y.L[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor] = R[\boxed{k(\underline{x})}^{\uparrow}, \lfloor y \rfloor]$ Although k appears on both sides, it's just there to indicate that rippling has failed to move the wavefront so that either side of the conclusion can be rewritten by the corresponding hypothesis expression. In practice both sides could have rippled partially, but not enough to permit weak fertilisation, and then the two wavefronts would be different. I shall show this more accurately below.

   Both sides are blocked. No cancellation is possible. It might be possible, in the case of a blocked ripple, to perform a tactic on the corresponding side of the hypothesis to produce a formula which would justify a rewrite of just the term which has been rippled past. This could be achieved by successively applying to both sides of the equality a function which is the inverse of the outermost function of a designated side. This can have the effect of raising a subterm of one side of the equality until it becomes one side of a new equality, thus "isolating" it.

   Whether or not such isolation would aid cancellation would depend on how deeply the blockage was nested. Take an extension of the schematic example:

$$\forall y'.L_1[L_2[x, y']] = R_1[R_2[x, y']] \vdash \forall y.L_1[\boxed{k_l(\underline{L_2[x, \lfloor y \rfloor]})}^{\uparrow}] = R_1[\boxed{k_r(\underline{R_2[x, \lfloor y \rfloor]})}^{\uparrow}]$$

   If the hypothesis could be "isolated" to give

$$\forall y'.L_2[x, y'] = L_1'(R_1[R_2[x, y']])$$

   we could rewrite the conclusion to

$$L_1[\boxed{k_l(\underline{L_1'(R_1[R_2[x, y]])})}^{\uparrow}] = R_1[\boxed{k_r(\underline{R_2[x, y]})}^{\uparrow}]$$

   But the chances of simplifying this down seem slight.

Rippling should ideally proceed as far as possible, so that the rewrite replaces as much of the original expression as possible. Currently weak fertilisation in CLAM applies when a whole side of an equality can be matched.

9.4.4 Assessing the Rippling Progress of Multiple Occurrences of the Induction Variable

It should be clear from this account that the combinations of possibilities are quite complicated, especially with multiple wavefronts. When reasoning about them, with a view to deciding when, where and how to generalise, identifying each occurrence of each variable separately is important so that it is possible to track:

- the subexpressions which must be identical and should therefore be treated alike. For example if we unfold the definition of multiplication to change $s(x) * y$ into $(x * y) + y$, we would want to know that the two y's are necessarily the same. Then if one is later identified, the other is too.

- the wavefronts, knowing which variable they originate from. This has two uses. If we decide that two variable instances are identical, then we know their corresponding wavefronts must be too. Conversely, if we find two wavefronts which are identical and have reached the same stage (e.g. rippled out), we may infer that the variable instances from which they originated are identical.

9.4.5 Induction Variables and Induction Schemes

Choosing an induction involves choosing a variable(s) on which induction is to be performed and an induction scheme. Schemes describe sets of base and step cases which must be proved to justify an overall proposition. The arrangement of base and step cases is itself the subject of proof that the cases they cover span the domain type of the variable they relate to. The proof will involve showing that the order in which the step cases traverse the domain is a *well-ordering*, so that the cases described do indeed step across the entire domain from the base case starting point(s).

We are familiar with some standard induction schemes - successor induction, list induction, course-of-values, etc. but there are infinitely many. Other schemes

are built from these. One way this may happen is explicitly and separately. Proofs of the validity of such composite schemes may be stored as lemmas.

An alternative way in which complex inductions can be achieved *indirectly* is through multiple inductions within a single proof, producing a composite induction across the whole proof. This has the advantages that the composite is built on demand, out of established schemes, and it needs no special proof of itself as a valid separate induction scheme. So when we do multiple inductions within a proof, they serve to create the combination of cases and induction hypotheses dynamically which could otherwise be reached through the definition of a fresh scheme. We can see this with the scheme for the *even* predicate, which steps in twos. If it is not available, two steps of successor induction achieve the same end.

So there is a significant relationship between the combination of inductions which take place in a proof, and the induction schemes available for proof construction. If composite induction schemes are explicitly available, a compact proof can result, using a smaller number of more powerful composite induction schemes. If composite induction schemes are not explicitly available, proof will have to resort to achieving their effects by combinations of weaker inductions, and the total number of inductions will be greater.

It is interesting to note that if a theorem as it stands is provable without generalisation, we never need to do more than one induction, provided we use a sufficiently powerful scheme. Clearly, we need to be able to create some subtle induction schemes. However if our range of available schemes is very broad, it is reasonable to use this information to suggest how many inductions should be required.

Since CLAM can create n-step inductions on demand, but has no facility for inductions based on more than one variable, it seems reasonable to assume as a heuristic that for a theorem provable without generalisation:

> *We should not need more inductions in any proof branch before some term cancellation is achieved, than the number of universally*

quantified variables present in the original problem. Those inductions should each act on variables which have not previously taken part in an induction.

Conversely, if we find that we are disobeying this heuristic, it suggests a reason to look for an alternative induction or try to generalise.

9.4.6 Effects of Using the Induction Hypothesis

How can we assess a use of an induction? We need to tell whether it brings us closer to completion, by ending or simplifying proof branches.

After longitudinal or transverse rippling, using the induction hypothesis, and cancelling, if the proof branch has not been completed, we face another proof subgoal. This may or may not still contain the induction variable or its derivatives (head, tail, predecessor etc.). We must then contemplate another induction proof. The next induction may be on a different variable from the set of variables present at the start of the problem. Alternatively, it may use a variable which has already taken part in an induction in order to create compositely an induction scheme not built into the system. In the latter case, the previous choice of induction scheme is called into question, as earlier discussion suggested. Since we have techniques for creating n-step inductions, the latter case should not be necessary, or at least we should be able to tell from the nature of the blockage that this is required.

The only inference rules which can complete the proof of an equality are the existence of an identical hypothesis, or the use of the reflexivity equality axioms. Functional substitution (cancellation) reduces the problem, by recursively demanding the equality of corresponding arguments.

If cancellation actually clears away terms corresponding to the pre-induction theorem, then, intuitively, a significant reduction of the problem has occurred. Other inferences such as rewriting and induction can only contribute to a situation in which one of these happens.

Successful longitudinal and transverse rippling create term structures which allow the induction hypothesis to be used to contribute to such simplifications.

This all suggests that an induction has proved worthwhile if it has achieved one of:

- a conclusion which is identical to the induction hypothesis;
- a conclusion true by the reflexivity of $=$;
- cancellation of at least some of the original term structure;
- a situation where subsequent induction on a different variable (from the set of variables originally present) leads to one of the above.

If this finite[2] process doesn't happen, induction isn't working, and generalisation should be explored to set up an induction which *will* work. The process should be guided by the failures of current inductions, and judged successful if it enables an induction which is worthwhile in the terms set out above.

9.5 Generalisation Guided by $\mathcal{MOR}$

Having addressed the problem of choosing when to generalise in section 9.4, in this section I shall investigate the questions of how to characterise a speculative generalisation in a flexible, expressive manner, and where to generalise, i.e. what terms to attempt to generalise.

I shall look at how induction can be patched first, and then see whether the kinds of unifications between induction hypothesis and conclusion ensuing will serve as justifications. Achieving unifiability will require computation in some cases, as noted in chapter 2. The need for justifiability is a useful restriction on speculations.

---

[2]Since there are only a finite number of variables in the original problem.

9.5.1 Patching Inductions

If we are to use generalisation to fix faltering inductions, we wish it to somehow clear up these errant wavefronts which are impeding matches with the hypothesis. There are a number of ways this can be achieved; we can either dispose of the wavefront or help it on its way.

We can describe each of these fixes speculatively in the same manner, as outlined at the beginning of this chapter:

- $\underline{\lambda u \lambda v.u}$

  $F_i(t, a) = t$, no generalisation.

- $\underline{\lambda u \lambda v.v}$

  $F_i(t, a) = a$, generalisation of this term to the new variable. It is thus distinguished, and may become the induction variable or a sink (or accumulator), as required.

- $\underline{\lambda u \lambda v.f(u, v)}$

  $F_i(t, a) = f(t, a)$, where f is some constant function symbol, or composition of such symbols. This permits the insertion of a sink (or accumulator) as in (9.6) of figure 9.1.

The higher-order unification required to identify the F_i is described in chapter 4, restricted as indicated earlier in this chapter. The F_i are function meta-variables.

To use MOR to describe a generalisation of the existing formula, we introduce a single new variable, a, say, to act as a sink or as an induction variable as required. Also, for each blockage point, where a wavefront has failed to ripple either into a sink or upwards out of the term, we introduce a separate function meta-variable F_i linking the new sink/induction variable to the blockage, i.e. a subterm, t, of the original will be replaced by $F_i(t, a)$ in the speculative generalisation. Placement of these meta-variable functions will be guided by a blocked proof, as I shall explain. Each inserted function meta-variable should

come between the blockage and wherever it is headed, i.e. around an outward bound wavefront, but inside an inward bound one, so it can create a sink.

Is this representation adequate to express any generalisation we might need to let an induction make progress at simplifying the theorem? For the generalisation of types 9.4-9.6 in figure 9.1 I believe so, for the following reasons. Our induction schemes currently apply to a single variable at a time. Therefore, this scheme should only need to create one new variable to be the induction variable. Might we need multiple *different* accumulators, as opposed to multiple copies of the same one? This is harder to say, but intuitively seems unlikely, since all the new variables introduced are universally quantified in the new theorem. If we are assuming single variable induction, then as the wavefronts are moved, we would then have to split them to make this necessary. That might have to be radically revised if we were wishing to use a tuple accumulator to amass multiple results from a function. Be that as it may, it works for a good range of problems.

Therefore, for a generalisation to enable an induction, the form of speculation permitted by the flexible meta-term I have described is adequate, unless the induction scheme is defined over more than one variable, in which case, I would need to have as many new variables as induction variables.

After inserting our speculation, we look to the proof process to guide us. The possibilities are:

- look at where the blockage occurs, and "get rid" of the whole term at that point, by turning it into a new, universally quantified variable. An example of when this is needed is for generalising:

$$\forall y \forall z.\ rev(t) <> (y <> z) = (rev(t) <> y) <> z$$
$$\vdash \forall y \forall z.\ \boxed{(\underline{rev(t)} <> h::nil)}^{\uparrow} <> (y <> z) =$$
$$(\boxed{(\underline{rev(t)} <> h::nil)}^{\uparrow} <> y) <> z$$

  This problem, which is handled by the kind of generalisation Boyer & Moore do, where multiplying occurring terms are generalised to new variables. The speculation would be:

$$\forall y \forall z \forall a.\ F_1(rev(t), a) <> (y <> z) =$$
$$(F_2(rev(t), a) <> y) <> z$$
$$\vdash \forall y \forall z \forall a.\ (F_1(\boxed{\underline{(rev(t)} <> h :: nil)}^{\uparrow}), a) <> (y <> z) =$$
$$(F_2(\boxed{\underline{(rev(t)} <> h :: nil)}^{\uparrow}, a) <> y) <> z$$

where both F_1 and F_2 will become instantiated to $\lambda u \lambda v.v$. The amenable generalised theorem is

$$\forall y \forall z \forall a.\ a <> (y <> z) = (a <> y) <> z$$

- get rid of a subterm of the blockage, by the same means. This could be just the variable occurrence that originated the blocked ripple. Here is such a problem just after induction:

$$t <> (t <> t) = (t <> t) <> t$$
$$\vdash \boxed{h :: \underline{t}}^{\uparrow} <> (\boxed{h :: \underline{t}}^{\uparrow} <> \boxed{h :: \underline{t}}^{\uparrow})$$
$$= (\boxed{h :: \underline{t}}^{\uparrow} <> \boxed{h :: \underline{t}}^{\uparrow}) <> \boxed{h :: \underline{t}}^{\uparrow}$$

This is the kind of problem which is solved by Aubin's technique of generalising variables apart. Here is a speculation which could achieve that:

$$\forall a.\ t <> (F_3(t, a) <> F_4(t, a)) = (t <> F_5(t, a)) <> F_6(t, a)$$
$$\vdash \forall a.\ t <> (F_3(\boxed{h :: \underline{t}}^{\uparrow}, a) <> F_4(\boxed{h :: \underline{t}}^{\uparrow}, a)) =$$
$$(t <> F_5(\boxed{h :: \underline{t}}^{\uparrow}, a)) <> F_6(\boxed{h :: \underline{t}}^{\uparrow}, a)$$

Only the occurrences which originate a blockage have been speculatively generalised. All the F_i turn out to be $\lambda u \lambda v.v$. The generalised theorem is

$$\forall x \forall a.\ x <> (a <> a) = (x <> a) <> a$$

- provide a sink, and redirect the wavefront into it. This is needed for the theorem which states that if you single step rotate a list as many times as the list is long, the result is the list you started with. The blocked proof looks like this:

$$rotate(len(t), t) = t$$
$$\vdash rotate(\boxed{s(\underline{len(t)})}^{\uparrow}, \boxed{h :: \underline{t}}^{\uparrow}) = \boxed{h :: \underline{t}}^{\uparrow}$$

A suitable speculation is:

$$\forall a.\ rotate(len(t), F_7(t, a)) = F_8(t, a)$$
$$\vdash rotate(\boxed{s(\underline{len(t)})}^{\uparrow}, F_7(\boxed{h :: \underline{t}}^{\uparrow}, a)) = F_8(\boxed{h :: \underline{t}}^{\uparrow}, a)$$

where F_7 is $\lambda u \lambda v.u <> v$ and F_8 is $\lambda u \lambda v.v <> u$

The generalised theorem is

$$\forall l \forall a.rotate(len(l), l <> a) = a <> l$$

- similarly, we might expect to be able to clear some longitudinal ripple's upward path using the same mechanism. The effect would be to substitute a nicer function or composition of functions, but the justification aspect would be harder. We'd have to show that $\forall x \exists y.old(x) = new(y)$, a non-trivial proof in itself. Since it is hard to see when this will be appropriate, I am not going to attempt this possibility.

A constraint becomes apparent. There is only a single new universally quantified variable, which appears in the same position in hypothesis and conclusion. The hypothesis copy may be instantiated to a different value from the conclusion one (as accumulators regularly are, as in tail-recursive synthesis). The hypothesis copy of this single new variable can only be instantiated once, to a single, common value. So once that value has been established, any speculation not consistent with it is pointless. This means that our new variable may only serve one purpose. We cannot simultaneously accommodate, say, transverse rippling onto an accumulator and turning a copy of a multiply-occurring variable into a simple non-induction variable. For example, take $x * x * x$, and consider speculating about it as

$$\forall a.\ F_1(x, a) * F_2(x, a) * F_3(x, a)$$

An attempt to identify

F_1 as $\lambda u \lambda v.u$,

F_2 as $\lambda u \lambda v.v$,

F_3 as $\lambda u \lambda v.u + v$

could not succeed, as there would be no consistent instantiation of the hypothesis copy of the new variable - one requiring it to also be a, the other requiring it to be a term strictly containing a. I shall refer to this requirement that the induction hypothesis be instantiable for the conclusion as the **rewritability constraint**.

It is important to note that these F's are meta-variables, not pure higher-order variables. Their identification is a matter for meta-level reasoning. As was the case for the instantiation solutions, restrictions like temporal scoping may be needed for the logic. Additionally, we may use knowledge about the proof to guide the instantiation process.

This technique is sufficiently expressive to cover most first-order generalisations. It is still further restricted, as I expect each F to be identified by matching with a single rewrite rule. Progressive matching first by one rule, leaving a partially instantiated function, and then by others until a whole match is achieved is difficult to control, and I do not attempt it.

9.5.2 Ensuring Justifiability

When will new formulae, adjusted by the effects of meta-functions, justify the originals? Schematically, we start with some

$$P[..., T_i[x], ..., T_j[x], ...]$$

where $T_i[x]$ and $T_j[x]$ are some distinguished subterms which will be considered for generalisation. The speculated generalisation is then

$$\forall a.P[..., F_i(T_i[x], a), ..., F_j(T_j[x], a), ...]$$

The justification proof would be of

$\forall a.P[..., F_i(T_i[x], a), ..., F_j(T_j[x], a), ...]$

$\vdash P[..., T_i[x], ..., T_j[x], ...]$

Each $F_i(T_i[x], a)$ could be:

- a projection onto its first argument. This restores the term structure, unchanged at that point, so justifiability is not an issue in relation to it.

- a projection onto its second argument, which is used as a non-induction variable or a new induction variable. Since the second argument is a new variable, universally quantified in the hypothesis, that can be instantiated to anything of the appropriate type, and justification is easy. However, since all the instances of a must be the same, this affects its use in the other generalisation speculations. There are two cases:

  - All the relevant $T_i[x]$ are equal. If they are not identical, computation must take place to make them so for instantiation to take place. Detecting and reconciling non-identical equalities is non-trivial. For heuristic purposes, we would almost certainly have to restrict this to use of existing lemmas.
  - They are not equal. The generalisation is wrong.

- F_i could be a function such that the wavefront can be rippled onto a acting as an accumulator. In this case we need there to be a **single** b such that **for all** the function meta-variables in a particular generalisation $\forall x.F_i(x, b) = x$, and then we know that this can be used to justify the original. The rewritability constraint can be seen to apply. This usage is incompatible with other usages of a in other function meta-variables, e.g. to define a new induction variable.

The order of the quantifiers is usually unproblematic since CLAM tries to re-order a sequence of universal quantifiers if it wants a later one at the front as the induction variable. This is usually possible for the kinds of re-ordering we

would want to make. It is not possible, for example, if it involves switching the quantification of a variable and the quantification of its type, i.e. we cannot switch the order of the following: $\forall \tau : \mathcal{U}\ \forall x : \tau \ldots$

From considering the requirement that the generalised theorem justify the original, we have another constraint on the generalisation. Since the new variable is used in the induction proof and will be instantiated to make an instance of the original formula, it must stand for the same term throughout, assuming justification will eventually take the form of instantiation. This is the **justification constraint**, another requirement relating to the new variable. Its demands sometimes subsume those of the rewritability constraint. It means, for example, that a commitment to generalising an unrewritten copy of a variable to a new variable is incompatible with use of accumulators. It is also incompatible with using the same new variable for generalising away a term which becomes blocked after any rippling has taken place, since the justification instantiation for that would be a strict superterm of the earlier one.

9.5.3 Controlling the Search for a Generalisation

Having selected a reason to generalise and a means of expressing a speculative generalisation, I must still have a way of picking the terms to mark out for potential generalisation.

There are various ways this could be done, the main choice is whether to start from where the blockage is manifested, or its originating variable. Either way, backtracking towards the other may be required, to consider generalising, respectively, smaller or larger terms. A further choice is whether to include all terms arising from copies of the induction variable in the speculation, or just the ones leading to blockages.

One solution is to start from the origins of the blockages, and see how their progress affects the proof. Backtracking in this case would be to progressively larger formulae.

Alternatively, I could look at the terms at which the blockage occurs and wrap the speculation around them in the original theorem, backtracking if necessary to subformulae. This could be a gross change to the formula, and runs a higher risk of generalisation to a non-theorem, which could be difficult to detect. It is attractive because it makes immediate use of the information about failure. It is important to avoid any implicit assumption that the original induction choice was correct. This may not be so, we may be trying to transfer the induction to the new variable.

Another approach would be to view the process as proof transformation rather than theorem transformation. Speculations would be introduced to progress the proof and then propagated back to the theorem. On failure, backtracking would roll the proof back and try a transformation at a previous stage. This would tend to follow the path of patching at the point of blockage, because each backtrack to a subterm would correspond to rolling the proof back to previous stages. An implicit assumption with this is that any wavefront which has apparently rippled out or into a sink is uncontroversial. This may not be so. A difficulty with this approach is that starting at the blockage can leave the new variable a long way from the origin it will have to propagate back to. A further problem is caused by having to decide which failed proof attempt to use for guidance. Fortunately there is little or no search with proof plans that succeed, so taking the first failure as initial guidance is reasonable. Inevitably there are almost always more failure states than starting states, as branching can occur after each step.

Since proof-patching motivates this technique for generalisation, and since starting from the original formula with guidance as to the origin of the blockages involves so many cases potentially, I have a system which selects terms at which blockage occurs in the first instance.

It is notable that this concentrates the generalisation on patching a particular proof attempt, when there may have been several proof attempts, by different uses of available lemmas. An aspect not dealt with here is the choice of which failed proof attempt to try and fix, if there is a choice from more than one.

The nature of the proof so far will also determine the type of generalisation attempted.

9.6 Implementation of MOR Generalisation

I have adapted the CLAM.v1.5 system so that it uses MOR to generalise theorems. My current implementation is lacking in some of the grander heuristics I described earlier to detect when induction was failing across nested inductions. It does, however, detect when proof attempts following a single induction fail to achieve fertilisation, and then tries to patch these by generalisation.

9.6.1 Deciding to Attempt Generalisation

The system takes failure to achieve strong fertilisation as an indication that induction might be improved. At that point, the failed proof is used to attempt to find a generalisation which will lead to either to strong fertilisation or to weak-fertilisation and cancellation. If generalisation fails, the theorem prover will continue without MOR, attempting to find a less successful weak fertilisation. If there is still no success, another attempt at MOR will try to find a generalisation which will lead to weak fertilisation.

This strategy of detecting failures and appealing to MOR to attempt generalisation are implemented adapting CLAM's overall induction strategy supermethod. The standard version ripples wavefronts out as far as possible, or into sinks, and then tries to fertilise, preferring strong fertilisation. The MOR system splits that standard version into two attempts. The first of these tries to achieve strong fertilisation or weak fertilisation with cancellation, as described above. The second only tries for the less successful kinds of fertilisation. They appear in CLAM's list of methods in that order, so that the stronger one will be tried first.

Immediately after each of the two versions (in CLAM's list of methods) comes its generalisation counterpart, which aims to complete the same proof stages as the corresponding induction strategy (induction, base cases, rippling, fertilisation etc.) but applied to a generalisation of the original. The counterpart also deals with a subproof justifying the original from the generalisation. The generalising supermethod has no immediate access to the failed proof attempt, but can reproduce it, by applying the same component methods to a step case up to the fertilisation stage. This provides the guidance for a speculation routine, which is described in the next section. The generalisation strategy, having picked a speculation, will pursue it through the rest of the standard strategy, using versions of the standard methods adapted for MOR.

There is some minor strategic adaptation here, so that deepest wavefronts are attempted first. This is to enable wavefronts requiring common generalisation to interact. Search is further controlled by having MOR select outward longitudinal rippling in preference to transverse rippling, and transverse rippling to inward longitudinal rippling which in turn is preferred to projection. This will become clearer from the description of speculation, and the example in the rotate proof, described later.

Failure results in backtracking to attempt a different speculation selection.

Failure of all the speculations resulting from this failed proof results in backtracking to attempt speculation from a version of the failed proof at an earlier proof stage, i.e. with fewer ripples.

9.6.2 Speculating about a Generalisation

Choosing speculations is driven by a need to make the changes where they are needed, not just a scattergun approach of changing everything in sight. The following analysis prefers not to try to change subterms, which although they may be blocked, may be perfectly alright in themselves, and only need something else to be generalised, i.e. they need a sink, or the other side of an equality to ripple successfully and enable cancellation.

Here is the algorithm for speculation for the system I have built.

1. Identify all the blockage points in the conclusion of the given failed proof.

2. Sort them, deepest first. There is no unique solution to this, but any which do fall below others can be considered earlier. The reason for doing this is so that speculations inspired by different blocked wavefronts may require a speculation to take place at a common place. This makes it easier to co-ordinate their sharing of a single speculation. Work on outward-bound wavefronts before inward-bound ones.

   For each blockage, check the following possibilities, and store all the relevant speculations in this order of preference:

   - Is the blockage adjacent to an equality? Such wavefronts may need no speculation, they may only need the other side of the equality to ripple accordingly. This gives us the option of not wasting effort on speculating about something which is not actually failing. The "speculation" is to leave the term unchanged.
   - Could this blockage be removed if a sink were available in another subterm position? I.e. is there a wave rule which would apply if a sink were available? If so, note the speculation which would potentially insert such a sink in the relevant position.
   - The last alternative, which always applies, is to speculate at the blockage point. This may lead to the speculation function being a projection onto one of its arguments, or something which enables longitudinal rippling. Strictly, this case subsumes the first one.

   Select one of the suggested speculations and apply it to hypotheses and conclusion of the blocked sequent. Backtracking may return for an alternative.

The following three sections discuss and demonstrate the MOR approach's working on some examples.

9.7 Generalising Variables Apart Using MOR

Here is an example of a theorem which is easy to prove if some of the variables are generalised apart.

$$\forall x.x + (x + x) = (x + x) + x$$

I shall assume that the only available wave rule is the step case of the definition of $+$:

$$\boxed{s(\underline{u})}^{\uparrow} + v \rightarrow \boxed{s(\underline{u + v})}^{\uparrow}$$

9.7.1 What Goes Wrong?

Although the theorem is "simpler" than the general version, the induction is made difficult by the occurrences of the variable in positions which can't be affected by wave rules:

$$x + (x + x) = (x + x) + x$$
$$\vdash \boxed{s(\underline{x})}^{\uparrow} + (\boxed{s(\underline{x})}^{\uparrow} + \boxed{s(\underline{x})}^{\uparrow}) = (\boxed{s(\underline{x})}^{\uparrow} + \boxed{s(\underline{x})}^{\uparrow}) + \boxed{s(\underline{x})}^{\uparrow}$$

The usual ripples apply:

$$x + (x + x) = (x + x) + x$$
$$\vdash \boxed{s(\underline{x + (\boxed{s(\underline{x})}^{\uparrow} + \boxed{s(\underline{x})}^{\uparrow})})}^{\uparrow} = \boxed{s(\underline{(x + \boxed{s(\underline{x})}^{\uparrow}) + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}$$

but neither side of this matches the induction hypothesis. We also get an extra ripple:

$$x + (x + x) = (x + x) + x$$
$$\vdash \boxed{s(\underline{x + \boxed{s(\underline{x + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}})}^{\uparrow} = \boxed{s(\underline{(x + \boxed{s(\underline{x})}^{\uparrow}) + \boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow}$$

but that doesn't help.

This attempt at induction can be seen to have failed because it doesn't even get as far as fertilisation, there is no other variable to do an induction on, and there are no rippling rules which might apply to the blockages, and thus act in concert to create a better induction scheme.

9.7.2 A Solution

The solution is to differentiate between the x's and generalise the theorem to

$$\forall x, \forall y. x + (y + y) = (x + y) + y$$

then the induction goes through:

$$\forall y. x + (y + y) = (x + y) + y)$$

$$\vdash \forall y. \boxed{s(\underline{x + (y + y)})}^{\uparrow} = \boxed{s(\underline{(x + y) + y})}^{\uparrow}$$

The tricky bit, of course, is working out how to choose the x's and y's. It now becomes clearer why Aubin's technique of defining *primary recursion paths* was sometimes effective. It was because this identified those occurrences which were likely to ripple outwards, since each ripple would produce a new term in the correct argument position for a further ripple. His analysis had two deficiencies:

- The new term might be needed in a position for which no definitional rule was available, consequently his algorithm would have no reason to generalise it.

- Wave rules in Aubin's terms were **only recursive definitions**. Although my only wave rule here is just a recursive definition, in principle I could have some which weren't.

Using Rippling and MOR

By labelling each variable and each wavefront by the occurrence of the variable it originated from, we can distinguish the individual ones and analyse their behaviour:

$$x_1 + (x_2 + x_3) = (x_4 + x_5) + x_6$$

$$\vdash \boxed{s(\underline{x_1})}^{\uparrow}_1 + (\boxed{s(\underline{x_2})}^{\uparrow}_2 + \boxed{s(\underline{x_3})}^{\uparrow}_3 = (\boxed{s(\underline{x_4})}^{\uparrow}_4 + \boxed{s(\underline{x_5})}^{\uparrow}_5) + \boxed{s(\underline{x_6})}^{\uparrow}_6)$$

the ripples can now be distinguished, this becomes:

$$x_1 + (x_2 + x_3) = (x_4 + x_5) + x_6$$

$$\vdash \boxed{s(\underline{x_1} + \boxed{s(\underline{x_2} + \boxed{s(\underline{x_3})}^{\uparrow}_3)}^{\uparrow}_2)}^{\uparrow}_1 = \boxed{s((\underline{x_4} + \boxed{s(\underline{x_5})}^{\uparrow}_5) + \boxed{s(\underline{x_6})}^{\uparrow}_6)}^{\uparrow}_4$$

Everything is longitudinal in this problem.

Informally, the lack of available ripples suggests that the third, fifth and sixth occurrences of x are non-induction variables. The first, second and fourth occurrences of x are possible induction candidates. The first and fourth occurrences ripple out completely, making them strong induction candidates. In terms of the analysis of induction hypothesis usage, longitudinal rippling can occur on both sides. There is no reason to look to replace the first and fourth occurrences as induction candidates. Our use of the speculation function will only be to find a way of generalising the rest of the term structure to make the current induction successful.

Inserting the speculative meta-functions turns this into:

$$\forall a.x_1 + F_2((x_2 + F_3(x_3, a)), a) = (x_4 + F_5(x_5, a)) + F_6(x_6, a)$$

$$\vdash \forall a.\boxed{s(\underline{x_1} + F_2(\boxed{s(\underline{x_2} + F_3(\boxed{s(\underline{x_3})}^{\uparrow}_3, a))}^{\uparrow}_2, a))}^{\uparrow}_1$$

$$= \boxed{s((\underline{x_4} + F_5(\boxed{s(\underline{x_5})}^{\uparrow}_5, a)) + F_6(\boxed{s(\underline{x_6})}^{\uparrow}_6, a))}^{\uparrow}_4$$

Looking at each of the outstanding wavefronts:

5 There is nothing that F_5 could be instantiated to, apart from $\lambda u \lambda v.v$ which will remove its blockage. We note that this will require the instantiation of a to x in the justification branch of the proof. The rewritability constraint means that such a generalisation is only compatible with other generalisations of x to a.

6 The identical argument holds for F_6, and its instantiation for a is consistent.

3 A similar argument holds for F_3, and its instantiation for a is consistent with those above, but it is complicated by being inside wavefront 2.

2 The term containing F_2 and F_3 is nested. According to the selection of F_2, F_3 may be irrelevant. There is no wave rule which could be made to ripple wavefront 2, so an alternative is to eliminate it by choosing F_2 to be $\lambda u \lambda v.v$, too. However, the justification branch instantiation is then $x + x$ for a. This is incompatible with the F_5 and F_6 generalisations.

The left-hand side of this equation can be made rewritable by using the last of these options, but it is then impossible to fix the right-hand side. The right-hand side can be made rewritable by using the first two of these options in tandem. Although this is inconsistent with the current version of adjusting the left-hand side, we can move back to the earlier, unrippled failed proof attempt. Exploring this latter course is preferable to soldiering on with a solution which can only be partial. We examine:

$$\forall a.x_1 + F_2^-((x_2 + F_3(x_3, a)), a) = (x_4 + a) + a$$

$$\vdash \forall a.\boxed{s(\underline{x_1})}^{\uparrow}_1 + (F_2^-(\boxed{s(\underline{x_2})}^{\uparrow}_2, a) + F_3(\boxed{s(\underline{x_3})}^{\uparrow}_3, a)) = (\boxed{s(\underline{x_4})}^{\uparrow}_4 + a) + a)$$

F_2^- denotes the version of F_2 at the previous, unrewritten stage. It is immediately apparent that the left-hand-side can be totally unblocked by making both F_2^- and F_3 $\lambda u \lambda v.v$. The outermost wavefronts are identical, and cancellation is enabled - our criteria are satisfied.

Trying this technique on more challenging problems like $\forall x.x * (x + x) = (x * x) + (x * x)$ shows up how difficult it is to tell when proofs are failing. Currently, I have some heuristics for this but they need refinement.

9.8 Generalisation by Adding Accumulators

In this example, we need to generalise to insert sinks, and to make the right-hand side work transversely, like the left. It is interesting in that a variety of longitudinal and transverse rippling has to take place, and the speculation functions involved take different values here, unlike in previous examples.

$$\forall l.rotate(len(l), l) = l$$

The typing of l as a list is elided. This is an example of a theorem which can be proved by adding an accumulator, and so generalised to:

$$\vdash \forall l \forall p.\ rotate(len(l), l <> p) = p <> l$$

where *rotate* and *len* are defined as

$$\begin{aligned} l \neq nil \rightarrow rotate(s(n), l) &= rotate(n, cdr(l) <> (car(l) :: nil)) \\ rotate(s(n), nil) &= nil \\ rotate(0, l) &= l \\ len(h :: t) &= s(len(t)) \\ len(nil) &= 0 \end{aligned}$$

car and *cdr* are just functions returning the head and tail of a list, respectively, and $<>$ is the append function:

$$nil <> l = l \tag{9.2}$$

$$(h :: t) <> l = h :: (t <> l) \tag{9.3}$$

9.8.1 What Goes Wrong?

List induction in the step case gives us an induction hypothesis:

$$rotate(len(t), t) = t$$

and the goal of proving:

$$\vdash rotate(len(\boxed{h :: \underline{t}}^{\uparrow}), \boxed{h :: \underline{t}}^{\uparrow}) = \boxed{h :: \underline{t}}^{\uparrow}$$

only one longitudinal ripple is available, making this

$$\vdash rotate(\boxed{s(\underline{len(t)})}^{\uparrow}, \boxed{h :: \underline{t}}^{\uparrow}) = \boxed{h :: \underline{t}}^{\uparrow} \tag{9.4}$$

and again the only possible transverse ripple produces:

$$\vdash rotate(len(t), \boxed{\underline{t} <> (h :: nil)}^{\downarrow}) = \boxed{h :: \underline{t}}^{\uparrow} \tag{9.5}$$

It is worth noting that CLAM would not currently permit this ripple, in the absence of a sink to ripple into. 9.5 is clearly not going to match the induction hypothesis on the left of the equality. Weak fertilisation is available on the right hand side:

$$\vdash rotate(len(t), \boxed{\underline{t} <> (h :: nil)}^{\downarrow}) = \boxed{h :: \underline{rotate(len(t), t)}}^{\downarrow} \tag{9.6}$$

but then we're stuck, and no cancellation is possible. The rippling on the left is blocked. The right hand side of the equality has rippled longitudinally but only transverse rippling is available on the left hand side, so there is a mismatch which my earlier analysis would reject. As I described in section 9.4, attempting weak fertilisation when one side of the equality is operating transversely, but the other is longitudinal is unlikely to lead to success in the form of a simpler goal to prove.

9.8.2 A Solution

This can be fixed by a generalisation of the original proof. We prove

$$\forall l \forall p.\ rotate(len(l), l <> p) = p <> l \tag{9.7}$$

Here's a proof of the step case:

$$\forall p.\ rotate(len(t), t <> p) = p <> t$$

$$\vdash \forall p.\ rotate(len(\boxed{h :: \underline{t}}^{\uparrow}), (\boxed{h :: \underline{t}}^{\uparrow}) <> \lfloor p \rfloor) = \lfloor p \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow})$$

As a universally quantified variable in the hypothesis, p is a potential sink. Assuming the wave rules in Table 9–1, two longitudinal ripples, L1 and L2, make the conclusion:

$$\vdash \forall p.\ rotate(\boxed{s(\underline{len(t)})}^{\uparrow}, \boxed{h :: \underline{(t <> \lfloor p \rfloor)}}^{\uparrow}) = \lfloor p \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow})$$

then, by T5, the definition of rotate, a transverse wave rule (the condition is trivial),

$$\vdash \forall p.\ rotate(len(t), \boxed{\underline{(t <> \lfloor p \rfloor)} <> h :: nil}^{\downarrow}) = \lfloor p \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow})$$

The final step needed on the left-hand side is to ripple the wavefront into the target, using the associativity of <> as a wave rule downwards:

$$\vdash \forall p.\ rotate(len(t), t <> (\boxed{\underline{\lfloor p <> (h :: nil) \rfloor}}^{\downarrow}) = \lfloor p \rfloor <> (\boxed{h :: \underline{t}}^{\uparrow})$$

And on the right hand side, using T1:

$$\vdash \forall p.\ rotate(len(t), t <> (\boxed{\underline{\lfloor p <> (h :: nil) \rfloor}}^{\downarrow})$$

$$= \boxed{(\underline{\lfloor p <> (h :: nil) \rfloor})}^{\downarrow} <> t$$

Now we can introduce p and fertilise with the induction hypothesis.

9.8.3 MOR on the Generalised Theorem

I shall return to the stage of the theorem where CLAM fails. Both sides of (9.4) are effectively blocked. The right hand side appears to ripple, but since its direction is mismatched against the left's, we don't accept it. We could be looking for an entirely new induction by generalising a term to a new variable or perhaps a patching of the existing proof.

$len(\boxed{H :: \underline{T}}^{\uparrow})$	$\rightarrow$	$\boxed{s(\underline{len(T)})}^{\uparrow}$	L1
$(\boxed{H :: \underline{T}}^{\uparrow}) <> L$	$\rightarrow$	$\boxed{H :: \underline{(T <> L)}}^{\uparrow}$	L2
$(\boxed{X <> \underline{Y}}^{\uparrow}) <> Z$	$\rightarrow$	$\boxed{X <> \underline{(Y <> Z)}}^{\uparrow}$	L3
$X <> (\boxed{\underline{Y} <> Z}^{\uparrow})$	$\rightarrow$	$\boxed{\underline{(X <> Y)} <> Z}^{\uparrow}$	L4
$L <> (\boxed{H :: \underline{T}}^{\uparrow})$	$\rightarrow$	$\boxed{\underline{(L <> H :: nil)}}^{\downarrow} <> T$	T1
$\boxed{\underline{(X <> H :: nil)}}^{\uparrow} <> Y$	$\rightarrow$	$X <> (\boxed{H :: \underline{Y}}^{\downarrow})$	T2
$(\boxed{\underline{X} <> Y}^{\uparrow}) <> Z$	$\rightarrow$	$X <> (\boxed{Y <> \underline{Z}}^{\downarrow})$	T3
$X <> (\boxed{Y <> \underline{Z}}^{\uparrow})$	$\rightarrow$	$(\boxed{\underline{X} <> \underline{Y}}^{\downarrow}) <> Z$	T4
$L \neq nil \rightarrow rotate(\boxed{s(\underline{N})}^{\uparrow}, L)$	$\rightarrow$	$rotate(N, \boxed{\underline{cdr(L)} <> (car(L) :: nil)}^{\downarrow})$	T5

Labels for longitudinal rules begin with L, and those for transverse rules with T.

Table 9–1: Wave rules used in the *rotate − length* theorem

Using the same strategy as before, we fit the blockages with speculation functions. Here the blockage is a longitudinally rising wavefront stuck inside a transverse function. We must either dispose of the wavefront or enable the transverse ripple to proceed by providing a sink. This could not suggest an alternative induction. Speculative insertion at earlier stages could, theoretically.

A question that arises here is the type of a. We would like the requirements of rippling to constrain that[3]. Unfortunately, Huet's algorithm cannot proceed without knowing the type in advance. Currently I guess this by choosing the predominant type of the formula, that of the equality itself.

We are now attempting to ripple transversely on both sides of the equality. Since the right-hand side is apparently rippled out, the first attempt would be to try:

$$\forall a'.\ rotate(len(t), F(t, a')) = t$$

$$\vdash \forall a.\ rotate(len(t), \boxed{F(\underline{t}, \underline{\lfloor a \rfloor}) <> (h :: nil)}^{\downarrow}) = \boxed{h :: \underline{t}}^{\uparrow}$$

[3]David Pym's unification algorithm or similar should be used for this, because Huet's needs to know the types.

No speculation can turn this into something which achieves strong fertilisation. That could only happen if *rotate* were defined longitudinally.

Backtracking finds another speculation:

$$\forall a'.\ rotate(len(t), F_1(t, a')) = F_2(t, a')$$
$$\vdash \forall a.\ rotate(\boxed{\underline{s(len(t))}}^{\uparrow}, F_1(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor a \rfloor)) = F_2(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor a \rfloor)$$

On the right-hand side, use of any longitudinal rule to instantiate F_2 will fail to reach strong fertilisation, for the same reason that not not generalising at all failed. However, wave rule T1 matches, instantiating F_2 to $\lambda u \lambda v.v <> u$. This will require a' to be $a <> (h :: nil)$. No other transverse wave rule matches, because no other transverse rule has the :: constructor as the main functor of the wavefront.

On the left-hand side, there are two blocked wavefronts. Taking the deepest one, in the second argument position of *rotate*, we seek a longitudinal wave rule. L1 is not applicable because the result must be a list. L2 is the only other available longitudinal rule. F_1 is instantiated to $\lambda u \lambda v.u <> v$:

$$\forall a'.\ rotate(len(t), t <> a') = a' <> t$$
$$\vdash \forall a.\ rotate(\boxed{\underline{s(len(t))}}^{\uparrow}, \boxed{h :: \underline{t <> \lfloor a \rfloor}}^{\uparrow}) = \left\lfloor \boxed{\underline{a <> h :: nil}}^{\downarrow} \right\rfloor <> t$$

Now, the last blocked wavefront is tried, and the definition of *rotate* applies:

$$\forall a'.\ rotate(len(t), t <> a') = a' <> t$$
$$\vdash \forall a.\ rotate(len(t), \boxed{\underline{t <> \lfloor a \rfloor <> h :: nil}}^{\downarrow}) = \left\lfloor \boxed{\underline{a <> h :: nil}}^{\downarrow} \right\rfloor <> t$$

lastly longitudinal rippling in downwards using L4 backwards (there is no alternative) moves the lingering wavefront down into the sink.

$$\forall a'.\ rotate(len(t), t <> a') = a' <> t$$
$$\vdash \forall a.\ rotate(len(t), t <> \left\lfloor \boxed{\underline{a <> h :: nil}}^{\downarrow} \right\rfloor) = \left\lfloor \boxed{\underline{a <> h :: nil}}^{\downarrow} \right\rfloor <> t$$

9.9 Generalisation of Terms

The major example used in Boyer & Moore's book [Boyer & Moore 79] is

$$\forall x \forall y.\ rev(rev(x) <> y :: nil) = y :: rev(rev(x))$$

where all functions are as specified before, and rev is defined as

$$\begin{aligned} rev(nil) &= nil \\ rev(h :: t) &= rev(t) <> (h :: nil) \end{aligned}$$

CLAM would use the definition to add to the wave rules in table 9–1, and I shall assume also the wave rules arising from the lemma relating rev and $<>$:

$rev(\boxed{H :: \underline{T}}^{\uparrow})$	$\rightarrow$	$\boxed{\underline{rev(T)} <> (H :: nil)}^{\uparrow}$	L5
$rev(\boxed{X <> \underline{Y}}^{\uparrow})$	$\rightarrow$	$\boxed{\underline{rev(Y)} <> rev(X)}^{\uparrow})$	L6
$rev(\boxed{\underline{X} <> Y}^{\uparrow})$	$\rightarrow$	$\boxed{rev(Y) <> \underline{rev(X)}}^{\uparrow})$	L7
$rev(\boxed{\underline{X} <> \underline{Y}}^{\uparrow})$	$\rightarrow$	$\boxed{\underline{rev(Y)} <> \underline{rev(X)}}^{\uparrow})$	L8

Recursion analysis would suggest induction on x, and the step case of the induction would look like this:

$$\forall y.\ rev(rev(t) <> y :: nil) = y :: rev(rev(t))$$

$$\vdash \forall y.\ rev(rev(\boxed{h :: \underline{t}}^{\uparrow}) <> \lfloor y \rfloor :: nil) = \lfloor y \rfloor :: rev(rev(\boxed{h :: \underline{t}}^{\uparrow}))$$

unfolding the definition of reverse makes this

$$\forall y.\ rev(rev(t) <> y :: nil) = y :: rev(rev(t))$$

$$\vdash \forall y.\ rev((\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}) <> \lfloor y \rfloor :: nil) =$$

$$\lfloor y \rfloor :: rev(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow})$$

A transverse ripple is available on the left hand side to get the wavefront towards the $y :: nil$ term. Generalisation might provide both sides with a new target.

$$\forall y.\ rev(rev(t) <> y :: nil) = y :: rev(rev(t))$$

$$\vdash \forall y.\ rev(rev(t)) <> \boxed{(h :: nil <> \underline{\lfloor y \rfloor :: nil})}^{\downarrow} =$$

$$\lfloor y \rfloor :: rev(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow})$$

The proof is stuck. Boyer & Moore's solution would generalise both $rev(x)$'s in the original theorem to a new variable, which would clear the blockage.

9.9.1 MOR on the Generalised Theorem

Attempt 1

Speculatively generalising produces:

$$\forall y \forall a.\ rev(rev(t) <> F_1(y :: nil, a)) = y :: rev(F_2(rev(t), a))$$

$$\vdash \forall y \forall a.\ rev(rev(t) <> \boxed{(h :: nil <> F_1(\underline{\lfloor y \rfloor :: nil}, \lfloor a \rfloor))}^{\downarrow}$$

$$= \lfloor y \rfloor :: rev(F_2(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor a \rfloor))$$

On the left, we might think that F_1 projecting onto its second argument could provide a list accumulator as opposed to a number accumulator, and requiring F_2 to provide a transverse ripple on the right. But the justification constraint demands an a^* such that

$$\forall z. F_1(z, a^*) = z = F_2(z, a^*)$$

So this kind of projection is impossible. Rippling is not available on the left. We must revert to an earlier proof stage.

Attempt 2a

We undo the last rewrite, (in the sense of keeping the number of rewrites on any term shortest) and try again:

$$\forall y \forall a.\ rev(F_1^-(rev(t), \lfloor a \rfloor) <> \lfloor y \rfloor :: nil) = \lfloor y \rfloor :: rev(F_2(rev(t), \lfloor a \rfloor))$$

$$\vdash \forall y \forall a.\ rev(F_1^-(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor a \rfloor) <> \lfloor y \rfloor :: nil)$$

$$= \lfloor y \rfloor :: rev(F_2(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor a \rfloor))$$

No longitudinal rippling is available on the left.

Both F_i could be $\lambda u \lambda v.u <> v$, and we ripple the wavefronts sideways onto a new accumulator using T2. The instantiated, rippled proof would be:

$$\forall y \forall a.\ rev((rev(t) <> a) <> y :: nil) = y :: rev(rev(t) <> a)$$

$$\vdash \forall y \forall a.\ rev((rev(t) <> \boxed{(\lfloor h :: nil <> a \rfloor)}^{\downarrow}) <> \lfloor y \rfloor :: nil)$$

$$= \lfloor y \rfloor :: rev(rev(t) <> \boxed{(\lfloor h :: nil <> a \rfloor)}^{\downarrow})$$

There is now no trouble in strong fertilising and cancelling this. Interestingly, it is not the solution Boyer & Moore find.

Attempt 2b

Looking again at the meta-level version of the sequent that started **Attempt 2a**:

$$\forall y \forall a.\ rev(F_1^-(rev(t), \lfloor a \rfloor) <> \lfloor y \rfloor :: nil) = \lfloor y \rfloor :: rev(F_2(rev(t), \lfloor a \rfloor))$$

$$\vdash \forall y \forall a.\ rev(F_1^-(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor a \rfloor) <> \lfloor y \rfloor :: nil)$$

$$= \lfloor y \rfloor :: rev(F_2(\boxed{\underline{rev(t)} <> h :: nil}^{\uparrow}, \lfloor a \rfloor))$$

The generalisation Boyer & Moore reach would be produced if both F_i project onto their second arguments, as new induction variables, and we propose a new

theorem. The justification constraint is satisfied, since a stands for the same term in each case. The generalisation proposed is:

$$\forall y \forall a. rev(a <> y :: nil) = y :: rev(a)$$

9.10 Conclusion

By taking failure of fertilisation (to either apply or to lead to cancellation of some of the original term structure) as a signal that an induction on the current theorem is inappropriate, we can choose to apply generalisation techniques precisely to address induction's problem. In order to detect this, we must work with supermethods, as freestanding components would be unable to monitor the history of the proof so far.

Using the existing induction strategy as guidance, a theorem can be coerced into a generalisation which effects some useful reduction on the theorem.

The following points form the basis of the algorithm:

1. Attempt generalisation when there is no available sequence of inductions which will lead to cancellation.

2. Analyse the combination of longitudinal and transverse rippling. This must include the progress in the proof so far, the dominant functors on either side of an equality, and the wave rules which might apply immediately beyond a blockage.

3. Interpose speculation meta-functions as described around the blocked terms, in such a way as to enable either the blocked ripple, or remove the blockage. Maintain a list of their justification constraints, and note whether alternatives might be available at an earlier proof stage.

4. If a new variable is inserted which is not fulfilling the rôle of a sink, resubmit the problem to recursion analysis, in case the new variable should take over as induction variable.

5. For each possible generalisation, according to the list of constraints, examine whether it will lead to a fertilisation and cancellation.

6. For each possible generalisation, examine whether it will justify the original theorem.

7. On backtracking to earlier proof states, undo first those rewrites which occurred on the longest rewrite path, since that should maximise the chance of keeping parallel terms in synchrony.

The analysis makes explicit constraints which are implicit in other generalisation systems, and uses them for meta-level reasoning. They are

- the number of new variables required to achieve a proof
- the requirements of the justification proof
- the requirements for enabling a sufficient match with the hypothesis to enable a rewriting.

In this chapter, I have explained how generalisation works and why it is often closely linked with induction proofs. This analysis of the connections between generalisation and induction is the basis for the technique I describe of using failed induction proofs to guide generalisation attempts. The technique relies heavily on existing accounts of "successful" induction proofs. MOR is used to guide choices of speculations in order to find such a successful solution. This approach takes in a number of different "types" of generalisation, some of which can be handled by existing techniques, but deals with them all within a single framework. It is effective for problems such as the rotate-length one described above, which other systems cannot manage, and where quite complex generalisations are required, based on a detailed analysis of the proof process.

Further types of generalisation, such as the generalisation of constants to variables should fit into an extension of this analysis.

Chapter 10

Comparison with Related Work - Generalisation

Some comparisons of $\mathcal{MOR}$ with related work on generalisation have been noted in passing in the previous chapter. Comparison will be drawn together and considered more systematically in this short chapter.

10.1 Generalising Terms to Variables

Boyer & Moore's type of generalisation takes selected repeated terms and generalises them to new variables. It is noteworthy that the motivation they give for performing this generalisation is close to that driving the $\mathcal{MOR}$ generalisation, it is an aid to making future induction work.

Boyer & Moore regard generalisation as tidying up after a previous induction to clear the way for subsequent inductions. Their view is that an induction may leave extraneous term structure around which has fulfilled its function. This may seem odd, but it arises quite naturally after weak fertilisation. Recalling my analysis in section 9.4, consider an example of an equality proof where one side ripples completely, permitting weak fertilisation, but the other side is blocked. Schematically, it will be something like this:

$$L_1(L_2(x)) = R(x)$$

$$\vdash L_1(\boxed{k_l(\underline{L_2(x)})}^{\uparrow}) = \boxed{k_r(\underline{R(x)})}^{\uparrow}$$

After weak fertilisation the conclusion is:

$$\vdash L_1(\boxed{k_l(\underline{L_2(x)})}^{\uparrow}) = k_r(L_1(L_2(x)))$$

We now have two identical $L_2(x)$ terms. $L_2(x)$ is just the portion of the term structure that the wavefront got through before it became blocked. As the expression stands, further induction on x would have to repeat that rippling, and would become blocked on the left-hand side, just as before. To avoid this, Boyer & Moore wish to generalise such terms to a new variable.

Achieving exactly such generalisation as I have just described is not straightforward in a system without wavefront annotation. It is also hard in Boyer & Moore's system because the theorem-proving components (equivalent to CLAM's methods) operate without communication or knowing quite how they are fitting in with the effects of other components contributing to the current proof. Their current "proof plan" information is embedded in the current state of the formula and the knowledge of which "method" is currently being applied.

Accordingly, Boyer & Moore have to take a roundabout route to find these terms. Before trying to apply any induction, they have a generalisation component which looks for repeated terms which occur on either side of an equality or in separate literals. This will find cases like the one I have described above, and tidy up the theorem for the induction to follow. However, it is not triggered by strong fertilisation failing.

The disadvantages of this are that their technique is not as closely tied to its purpose as one would like, so it can easily be confused by multiply occurring terms which have not arisen from a previous fertilisation, such as:

$$\forall l.\ l <> (l <> l) = (l <> l) <> l \tag{10.1}$$

The dangers of false generalisations like this are considerable.

To avoid these unwise generalisations, their essential technique is hedged about by special purpose generalisation lemmas and extra conditions. The lat-

ter, listed in chapter 2, are to stop Boyer & Moore's system from generalising constants, and to make restrictions on the type of the generalised expression.

The analysis above suggests that CLAM should attempt generalisation of the terms corresponding to $L_2(x)$ after weak fertilisation, as part of the overall induction strategy. However, it is not enough to only consider such generalisation after weak fertilisation, because such a subgoal may be proposed as a theorem in its own right, and we need heuristics to be able to cope with it.

MOR generalisation returns to the original motivation for generalisation, of enabling successful induction, and uses CLAM's tools to construct just such a generalisation as will achieve a successful induction. It has both the wavefront annotation and the control of linking methods into plans to help it do this.

In cases like the one described above, MOR generalisation will discover that $L_2(x)$ is the term that needs to be generalised because rippling will be blocked there. So it will propose generalisation of the term that lead to weak rather than strong fertilisation taking place. So it ties in closely with the method which has just preceded it, even if they are not explicitly linked in a supermethod. Given any repeated terms which are obstructing the rippling of an induction step case, the MOR technique will be able to propose generalising them. It will fail to find some generalisations Boyer and Moore would find, if the term to be generalised is not obstructing rippling, as in this case:

$$\forall l \forall m \forall n.\ l <> (m <> reverse(n)) = (l <> m) <> reverse(n)$$

However, this theorem can be proved without generalisation by both CLAM and Boyer & Moore's system.

MOR generalisation will not propose all the generalisations that Boyer & Moore's heuristic would, because it is more conservative in the sense that it will not propose repeated terms *unless* they are obstructing rippling. So given 10.1 it will not propose a generalisation to the non-theorem:

$$\forall l \forall a.\ l <> a = a <> l$$

Instead it will find a solution in which the first occurrence of l on each side is distinguished.

10.1.1 Generalising Variables Apart

Generalising variables apart is a type of problem addressed by Raymond Aubin. Again, his system looks at the current formula to be proved, rather than the whole proof process. His system operated along the same lines as the Boyer & Moore theorem prover. In his case, however, there is a more explicit attempt to anticipate the subsequent proof.

Recognising that fertilisation was the key to success in induction, Aubin noted that the means of achieving it lay in having sequences of rewrites which would take the extra term structure introduced by induction (the wavefront in CLAM terminology, although Aubin had no wavefront notation), and move it upwards in the term, leaving a copy of the original for fertilisation. In CLAM's terms, this is rippling out. His system had only recursive function definitions to act as rewrite rules, not CLAM's more general notion of wave rules. Therefore, it was possible to analyse the formula from the top, and see the routes through it where there was the potential to move a wavefront all the way from the bottom up. For example, in:

$$\forall x.\ f_{[1]}(f_{[1,1]}(x), f_{[1,2]}(x)) = f_{[2]}(f_{[2,1]}(x), f_{[2,2]}(x))$$

if we assume that all f_i are defined on their first argument position, then there are only rewrite rules to move wavefronts occurring in the positions [1,1,1] and [2,1,1]. These were called *primary recursion paths*. The rewriting of any wavefronts occurring in non-primary recursive positions would become blocked.

This recursion path analysis was used to suggest generalising occurrences of the induction variable which were not on primary recursion paths, to some new non-induction variable which would not get in the way. The above would become

$$\forall x \forall y.\ f_{[1]}(f_{[1,1]}(x), f_{[1,2]}(y)) = f_{[2]}(f_{[2,1]}(x), f_{[2,2]}(y))$$

In practice, this characterisation was too rigid. In the case of the distributivity of multiplication over addition, a purely primary recursion path analysis would suggest generalising:

$$\forall x.\ x * (x + x) = (x * x) + (x * x)$$

to the non-theorem

$$\forall x \forall y.\ x * (y + y) = (x * y) + (y * y)$$

assuming $*$ and $+$ are recursively defined on their first arguments.

Consequently, Aubin extended his algorithm in two ways. Firstly, he added a module to try out some real values in the proposed generalisation, so that non-theorems could be rejected. Secondly, he allowed other permutations of the partitioning of the occurrences of the induction variable into induction and non-induction occurrences. Preference was given, not only to the occurrences on primary recursion paths, but if necessary to occurrences in a recursion argument, but not on a primary recursion path.

Aubin's technique amounts to a kind of speculation which tries out the values most likely to succeed. His analysis looks forward to the subsequent proof stages more than Boyer and Moore's.

In CLAM's terms, Aubin's approach finds the positions amenable to wave rules, casts them all as candidates for induction. Failure results in broadening the net, guided by some of trial and error. The notion of a primary recursion path was effectively an attempt at a heuristic which would stand a good chance of producing successful ripples.

Here again, making induction succeed is the motivation and the guide for generalisation. In this case analysis of the subsequent proof is used more directly to suggest suitable generalisations. Again, however, generalisation is a preprocessing step applied in order to make an induction succeed, not one applied in response to a failure to achieve fertilisation.

MOR generalisation is similarly motivated to Aubin's technique. It wishes to make a subsequent induction work, and takes advantage of an analysis of recursion to do so. The differences are:

- MOR generalisation is initiated in response to induction failing to lead to fertilisation. Aubin's like Boyer & Moore's, is a preprocessing step. It tries to generalise whether it needs to or not.

- The MOR approach is guided by a failed proof attempt, it does not analyse all the route ahead, as Aubin does.

- The analyses which lead to the proposals as to which terms should be generalised have similar origins, but different results, as I have described, since the MOR/CLAM system must handle a wider notion of rippling. MOR will try to generalise whatever obstructs rippling, not just repeated terms, as Aubin does. The result of MOR generalisation may insert sinks, since the analytical tools are available to do this, not just distinguishing induction and non-induction variables.

- Rather than explicitly computing the course of a future proof, as Aubin tries to do, MOR can embark on a proof and make a generalisation to fit in with proof requirements. In this respect, MOR is dynamically guided by being embedded in a proof plan supermethod. This gives the proof attempt an overall structure, but adapts to each stage of the proof.

- The MOR approach can take failure information from other sources - namely failure of a base case and continued failure to achieve fertilisation. The latter may be used to reject speculated generalisations until a generalisation is found which enables fertilisation.

10.2 Generalising Constants to Variables

Aubin had a further generalisation technique intended to generalise constants to variables, as described in 2.3.3. He used this in cases dealing with transversely defined functions, like the tail-recursive reverse in chapter 7. Suppose we have such a tail-recursive reverse, $reverse_{tr}$, and a naïve reverse, $reverse$, and we wish to prove

$$\vdash \forall l.\ reverse_{tr}(l, nil) = reverse(l)$$

After induction, the step case is

$$reverse_{tr}(t, nil) = reverse(t)$$

$$\vdash reverse_{tr}(\boxed{h :: \underline{t}}^{\uparrow}, nil) = reverse(\boxed{h :: \underline{t}}^{\uparrow})$$

Ignoring CLAM's insistence on the existence of sinks before transverse rippling, This evaluates to

$$reverse_{tr}(t, nil) = reverse(t)$$

$$\vdash reverse_{tr}(t, \boxed{h :: \underline{nil}}^{\downarrow}) = \boxed{append(\underline{reverse(t)}, h :: nil)}^{\uparrow}$$

The left-hand side is blocked. Weak fertilisation on the right makes this

$$reverse_{tr}(t, nil) = reverse(t)$$

$$\vdash reverse_{tr}(t, \boxed{h :: \underline{nil}}^{\downarrow}) = \boxed{append(\underline{reverse_{tr}(t, nil)}, h :: nil)}^{\uparrow}$$

This is badly stuck. The theorem can be proved by generalising it to

$$\vdash \forall l \forall a.\ reverse_{tr}(l, a) = append(reverse(l), a)$$

Here, the *nil* on the left has been replaced by an accumulator, which has also been added on the right.

In theorems stated with a constant in that accumulator position, the proof may be hard to achieve unless that the constant is generalised to a variable, otherwise fertilisation is obstructed. The MOR approach should be able to do this kind of generalisation too, but I haven't yet tried such an example. I believe it would find the generalisation proposed above.

10.3 Conclusion

Aubin's and Boyer & Moore's approaches always took multiple occurrences of the same term, and generalised them to a new variable. This meant that they had no possibility of generating a proposal which was not a generalisation of the original. In the MOR system, the generalisations proposed may be more subtle, and justification becomes an explicit issue. However, since it is also an explicit proof branch, unlike in the Boyer & Moore system, failure to achieve justification will cause backtracking to attempt an alternative generalisation.

The danger of the MOR approach leading to non-theorems is less than that in the Boyer & Moore case, but perhaps a little more than in Aubin's case. The extra protection of trying out some values cannot be denied. However, my analysis of successful induction in chapter 9 shows that there are numerous mechanisms for detecting failure, and therefore rejecting generalisations. I have demonstrated the effectiveness of this in my examples.

The MOR approach to generalisation can be seen to be well-motivated and subsume existing approaches to generalisation in the context of induction proofs. As I have shown in chapter 9, it covers examples these existing approaches cannot attempt, such as the rotate-length theorem. In summary, the reasons for this are:

- It is explicitly guided by a failed proof,
- It has the CLAM tools enabling characterisation of wavefronts, sinks and transverse movement, and
- Instead of being restricted to replacing repeated terms by a new variable, it can add sinks linked to the existing term structure by different functions.

Chapter 11

Conclusions and Further Work

11.1 Conclusions

This thesis examines and demonstrates the technique of $\mathcal{MOR}$ in the context of two classes of problem. In each of them it successfully expresses and develops the speculation one might expect of a human. This speculation is inspired by knowledge about proof structures, and afterwards constrained by other such knowledge, until a solution is reached. These two stages, of speculating and resolving the speculation require careful co-ordination.

The work on tail-recursive optimisation successfully exploits the proofs-as-programs paradigm to gain and harness new insights into tail-recursion. Systematically viewing generalisation as an aid to inductive theorem proving enables a number of forms of generalisation to be dealt with uniformly.

There is a valuable interplay between the automation and proof structure design. In order to use $\mathcal{MOR}$ at all, one must be very specific about exactly what rôle it is to play. Using a sensitive tool for exploring proofs, requiring fine control, has enabled us to make our planning more precise, so this work has contributed to our growing understanding of how proof works.

In this chapter I shall draw conclusions about

- the value and rôle of MOR,
- contributions to tail-recursive optimisation,
- contributions to generalisation,
- psychological validity.

Lastly, I will describe further work which could be undertaken.

11.1.1 MOR and Meta-Level Reasoning

MOR is used here specifically as an adjunct to meta-level reasoning, and can be seen to be effective precisely because the object/meta-level separation permits reasoning *about* the object level, and the postponement of object level constraints.

It seems similar to Prolog-style backwards reasoning with variables becoming instantiated as required. Superficially, that was almost certainly the technique's inspiration. It would be wrong to view MOR as a sequent calculus equivalent of Prolog. There is a significant qualitative difference in having an explicit meta-level. Prolog-style backwards reasoning alone could only offer limited MOR, not just because it is first-order, but because it is only object level. The additional use of meta-level control is a key factor in the success of MOR.

It is not surprising that this powerful technique requires considerable control. The key has been seen to be detailed analysis of proof structures, and the ability to describe components in such a way as to permit their flexible, goal-directed assembly, guided by knowledge of the theory. It has enabled a clear step in the direction of making plans and proof structure take over from logical syntax as the driver of proof production.

The flexibility of meta-level reasoning is necessary for precise control over MOR.

Meta-Variables

It is clear that the meta-variables used for *MOR* are subject to more constraints than ordinary variables of the relevant type would be. They are different from simply using higher-order object-variables. These constraints are:

- Temporal scoping - knowledge about which variables only came into existence after whatever the value of the meta-variable is, and could not, therefore, be part of its definition,
- The purpose for which the variables were created. By insisting that they fulfill certain rôles in strategies, we restrict the unifications we attempt to suit.

Unification alone is not enough to identify their values. Either the unifications chosen must be very carefully structured so as not to allow evasion of these constraints, or filtering must be performed to enforce them.

Inhibiting Multiple *MOR*

Once *MOR* is being attempted, and meta-variables have been introduced, all other methods must take account of that.

From the meta-level control point of view, rampant speculation is a hard thing to tame. Humans usually restrain themselves unless they're very confident of what they're doing. Looking at "The Big Picture" takes considerable depth and breadth of understanding. Our level of control of proofs is not yet up to managing extensive simultaneous speculation involving meta-variables.

Further, at the level of intricacy which would be involved, our higher-order unification tools can't cope. Higher-order unification will not retrieve us from a large-scale lack of precision involving many variables. Applied to terms containing multiple meta-variables in contiguous positions, it produces numerous flexible answers, which offer us little information. Indeed they can lead to an explosion of variables and unwanted speculation. That is why I make such efforts in

both of my problem areas to pin down the use of speculation and meta-variables very closely.

These problems are really two sides of the same coin.

The straightforward solution, which I have taken, is to insist that any method which would introduce meta-variables instigated by a new speculation should be inhibited from doing so if any meta-variables are already present. This kind of self-consciousness may seem rigid, but it amounts to insisting on only one speculation at a time. In 11.2.2 I sketch out a problem which would require multiple speculation.

Method and Supermethod

Methods designed to deal with a particular situation have the advantage of independent self-contained operation, mediated by their pre- and post-conditions. This works well for problems where the various stages *are* reasonably self-contained, or order of application is unimportant. In practice, even for its separate methods, CLAM conveys meta-level information from one method to another by embedding control information (wavefronts) in the object-level expression.

The same free-standingness may also be a disadvantage. Methods may be isolated and too general purpose to play a rôle required in a proof strategy. It is only within the context of a particular speculation that one knows what to expect or reject according to its needs. One may need to carry meta-knowledge about purpose across individual methods. A "supermethod" embodying an strategy constructed from ordinary methods and perhaps employing some specially written ones may be necessary. This is especially true of *MOR*, where we expect to know what kind of operations will provide the identity of the meta-variable.

The danger is that a method will apply to a meta-variable inappropriately, and apparent success will take the system down a blind alley. The errors here are of commission more than omission, although both are search problems. This is a problem wherever the existential method might apply, particularly. As it simply inserts a meta-variable for an existentially quantified variable, many symbolic

evaluation and tautology results supply possible values immediately, without giving a general strategy a chance to apply.

If, on the other hand, all the general purpose methods intended to function free-standing on a wide range of problems are adapted for particular ones, they will be made too specific and then not apply when they should more generally.

Propagation of Reasoning

One of the design decisions made in this implementation of MOR was to implement meta-variables as Prolog variables, for the purpose of propagating their values, and to give us easy access to first-order unification when that was appropriate. Mostly, Prolog's unification was intercepted and replaced by higher-order unification under the control of MOR. The choice of Prolog variables was natural in a Prolog setting. It ensured that any discoveries relating to the identity of the variable object were automatically propagated by Prolog's instantiation. There was no need to exchange new information for tokens throughout in all copies of the representation actively. The linguistic consequences of this decision are addressed below.

This confirms our intention that the speculative phase of MOR be a combination of instantiation and the creation of subgoals for any object-level requirements as they became manifest. The discovery phase was then achieved through proof, and its findings propagated consistently throughout, as they were found.

No mechanism was established for storing other requirements of the meta-variable's value. As anticipated this was not required, being handled by the means already described and embedded in the meta-level proof structure knowledge.

Representation and Language

The object-level language available for describing the problem will affect any use of MOR. It is necessary to reach an accommodation between the object language, the meta-level language and the implementation language.

To reason middle-out, we start with the object level language, and extend it by allowing meta-variables. This will inevitably break the grammar rules of that language, which enforce constructs that may well be the ones we wish to reason about. Exactly how they do this will influence the ease of adaptation of the representation. Meta-level languages are commonly more loosely defined and extensible, but assumptions may be built into them implicitly, in choices of representation. The underlying implementation language may have its own requirements as well.

An example is the Oyster system's notation for the application of a function, f, to an argument, a, is $f\ of\ a$. For those of us used to other styles, there is syntactic sugar to permit the $f(a)$ formulation. As Oyster is a higher-order system, f may legitimately be a variable, provided that it is appropriately typed and quantified. In either of these formulations, although illegal under the grammar of Martin-Löf Type Theory, it is perfectly possible to replace f by an untyped unquantified meta-variable, F, at the meta-level. However, as the system is built on Prolog, which is first-order, using the Prolog term structure to represent the term $F(a)$ would produce syntax errors, if Prolog variables were used to represent meta-variables. The basic Oyster $f\ of\ a$ representation was easy to adapt, and had no inadvertent interaction with the Prolog's term structure.

One of the original motivations for building CLAM was not just as a planning system, but as a way of avoiding some of the details of the object level until they proved necessary for a proof whose major components had been established. It was a tool for meta-level reasoning. Consequently, it was expressly designed to omit certain linguistic requirements of any underlying logic. Unfortunately some first-order assumptions were implicitly built in by using Prolog's term structure.

So here, we have relatively little difficulty in extending the underlying object level language, but are somewhat impeded by the meta-level language's assumptions.

Implementation Issues and Interface with Higher-Order Unification Subsystem

In spite of using Prolog variables for meta-variables in CLAM, the higher-order unification subsystem uses ground objects to represent higher-order variables, and assumed terms which were $\beta-$ and $\eta-$normal formed, and for which the type of all symbols was known. An interface was built to effect all this, and then instantiate the Prolog meta-variables when appropriate. This meant that the higher-order system could be kept localised, and the advantages of Prolog's backtracking and instantiation propagation could be used.

This could deviate from CLAM's choice of search strategy, in that the unification subsystem presented a list of alternative solutions from which choices were made one at a time, and backtracked over, essentially depth-first. To observe the search strategy would have meant drawing on this list of possibilities in the same manner as the current strategy, a desirable, but not particularly easy thing to do. As it would have made little difference to the information resulting from the experiment, no effort was made to implement this.

Normalisation

Types of normalisation arose from two sources. Huet's algorithm assumes expressions in $\beta-$ and $\eta-$ normal form. Oyster permits a syntactic sugar for application of a function to arguments which actually corresponds to an underlying representation of curried expressions or compound products. Both of these entail pre-processing steps for the interface into the unification subsystem. There is no great difficulty in achieving this.

The use of rewrite rules may result in the "output" of MOR being normalised. When this is put in a conclusion, for example, problems can appear if other expressions, such as a corresponding induction hypothesis, have not been similarly normalised. It may become difficult to see matches which were implicitly kept in step before. So these normalisations must be applied to all expressions involved

in matching. If other expressions, such as a corresponding induction hypothesis have not been similarly normalised, ordinary first-order unification may now fail.

This means that the use of the higher-order unifiability algorithm is necessarily pervasive through the system. Once its use has been initiated in relation to any particular data structure, it must be used from then onwards.

Types

The implementation of Huet's unifiability algorithm is now available as the basis for future work. It would be useful to extend it to deal with types more flexibly, for example as David Pym describes in his thesis [Pym 90]. He shows how one can start with variables whose types are also variable, and unification can instantiate the types as necessary. My implementation is currently very reliant on being able to guess the types of all the entities in a formula in advance of any unification, a hard task. Operation with variable types would be a helpful enhancement, allowing some of the guessing to happen at easier times. It would be a more natural way of reasoning middle-out, and would actually be essential to allow fuller MOR, as described below.

Failing that, an improved type-guessing algorithm should be produced. Mine is already far more adventurous than CLAM's, variously exploring terms top down and bottom up until it pieces together all the information without leaving any variable holes.

Search Control and Meta-Level Reasoning

Although conventional search control techniques were used, they were not always adequate to the purpose and the fit was sometimes contrived. In order to ensure that CLAM was always working on the next goal which was likely to prove informative, it became necessary to switch the order in which some of its methods produced subgoals, to place the informative ones first. This was an easy substitute for a more subtle and elaborate search control mechanism which

could have selected a subgoal according to some heuristic criterion. This is a complex problem to which I shall return in section 11.2, on further work.

11.1.2 Contribution to Tail-Recursive Optimisation

Central to this part of the research was the description of the necessary proof structure for tail-recursion to obtain in the functional extract term. Stan Wainer is to be thanked for his account of this. This characterisation of tail-recursive optimisation by the way the induction hypothesis was used provided a structure within which speculation could take place and be resolved. Importantly, it defined a purpose, a general goal to which efforts could be directed, unlike the static templates used elsewhere, or emulated indirectly procedurally. Tail-recursiveness was inherent to this proof structure, not an ancillary fact to be proved. Likewise, justification of the equivalence of the new program to its specification was a companion proof branch which only needed to be proved for the choices made in the individual example, not for a whole template. Depending on which lemmas were made available to the system, different proofs and therefore different programs could ensue.

The original theorem acted purely as a specification of the program. It was not used as an object to be transformed.

The middle-out system was able to synthesise tail-recursive versions of several standard naïvely defined functions: *reverse*, *length*, *times*, *greatest* and *total*. Additionally, it worked on two examples, each of which could elude some existing technique:

- Summation. The functional *sigma*0 was defined as the summation of a function's values from 0 up to a given value:

$$sigma0(f,n) == \sum_{i=0}^{n} f(i)$$

  Since this involves a functional, it would not be admissible in Huet and Lang's second-order system. Although summation from 0 seems restricted,

it allows the functional to be total, and defined using one of Oyster's basic types.

- Integer Half. This was defined using recursion stepping two-at-a-time, as for the definition of *even*:

$$\begin{aligned} multhalf(0,y) &== 0 \\ multhalf(s(0),y) &== 0 \\ multhalf(s(s(x)),y) &== plus(multhalf(x,y),y) \end{aligned}$$

This function would evade Darlington's F-matching because the step case requires matching a single template variable to a composite function $\lambda n.s(s(n))$

11.1.3 Contribution to Generalisation

By using the production of an inductive proof as motivation and guide for generalisation it has been possible to develop a single unified approach to deal with various generalisation problems previously treated separately or not at all.

The resulting generalisations are minimal, but this is an advantage when safeguarding against over-generalisation. The generalisation from

$$\forall x.x + (x + x) = (x + x) + x$$

will be found as

$$\forall x \forall y.x + (y + y) = (x + y) + y$$

not

$$\forall x \forall y \forall z.x + (y + z) = (x + y) + z$$

but I know of no theorem prover that can find this last generalisation in a principled way, i.e. other than with just trial and error.

Analysis of successful proof structures again provided the key to this task. It suggested that tracking individual sources of failure could suggest locations for potential generalisation, to eliminate or fix these obstacles. The precise generalisation needed would be discovered by the requirements of proof. Failed proof attempts provided initial information for the process. MOR was then used to insert meta-variables permitting speculation at the points of previous failure. Further MOR in the context of a subsequent proof attempt instantiated them to identify the actual generalisation needed for success.

As for the tail-recursive optimisation problems, this necessitated a detailed analysis of what constitutes "success" in inductive theorem proving, what forms impediments take, and how known generalisations remedy them.

It was surprising to find that the additions needed for the generalisation and its discovery could be restricted so compactly, normally to a single new variable and a functional meta-variable for each errant wavefront.

11.1.4 Psychological Modelling

One of the claims I made in my introduction for this technique was that of some psychological validity. In the light of the descriptions in this thesis, that should now be assessed. Psychological studies of the acquisition and practice of mathematical skills in humans propose that key abilities are those of classification and abstraction of structure. Using MOR for mathematics uses the same kinds of underlying mechanism - classes and structures - for the same sorts of purpose, in the same way that reports of mathematicians suggest they do - hypothesising guided by structures.

Using Classes

Piaget argues that one of the early abstract abilities children acquire is that of being able to recognise a *class* of entities [Piaget 52]. They learn to distinguish kinds of things - cups, balls, spoons, cuddles etc. This is first manifested by observation and manipulation of physical objects. It is further evinced by creation

of such objects, literally or in drawings. Children may use their classifications to request an object of the class, and accept what they are given according as whether it fits the purpose.

Class is a fundamental logical-mathematical concept and leads to the refinement of types of comparison:

> Is this a member of a class C, or is it not?
>
> Is this set of members of class C here bigger than that one there?
>
> Is this set of members of class C here bigger than that one of class D there?

The notion of class precedes those of quantity and number. This account is taken from Howard Gardner's book on theories of multiple intelligences [Gardner 84]. Gardner argues that hypothesising is an advanced activity, and symbolic reasoning comes much later, if at all.

As mathematicians, and creators of artificial mathematicians, we recognise classes of proofs, and identify classes of inference pattern which go into the making of proofs. We categorise the classes of objects involved. We build precise and powerful descriptions of these classes.

Having described mathematical classes, mathematicians can reason in terms of them, hypothesise about their members, determine whether given objects fall into certain classes, and sometimes even generate objects of a particular class. The technique of MOR corresponds closely to these abilities. It hypothesises new variables to fulfill the rôles such as that of an accumulator, or a function which will enable an accumulator to be used. In doing so, it uses knowledge about the existence of such classes of object.

Mathematicians also study all these classes' relationships to each other. The relationships between classes form structures, which are the subject of the next part of my argument.

Using Structures

The other essential component of MOR is its use of recognised structures for guidance. On this, Gardner cites the thoughts of Henri Poincaré, a famous mathematician from the 19th century, on the importance of structure, wondering: " why, if mathematics only involves the rules of logic, ... anyone should have difficulty in understanding mathematics?". Poincaré distinguished between prodigious memory and reasoning, believing that structured reasoning was the core mathematician's ability. He wrote (my emphasis):

> "A mathematical demonstration is not simply a single juxtaposition of syllogisms, it is syllogisms placed in a certain order, and the order in which these elements are placed is much more important than the elements themselves. If I have the feeling, the intuition, so to speak, of this order, so as to perceive at a glance the reasoning as a whole, I need no longer fear lest I forget one of the elements, for *each of them will take its allotted place in the array*, and that without any effort of memory on my part."

This approaches a description of MOR. A later quotation about mathematicians is evocative, they "are guided by intuition and, at the first stroke, make quick, but sometimes precarious conquests, like bold cavalry men of the advance guard".

The value of structures is respected in other psychological literature, too. In "The Psychology of Learning Mathematics" [Skemp 71], Skemp places great emphasis on the *schema* - "the general psychological term for a mental structure". A schema "integrates existing knowledge, and it is a mental tool for the acquisition of new knowledge". He describes striking results from experiments to compare rote learning and schematic learning, in which students were able to recall twice as much schematically learnt material as rote learnt. Four weeks later, the ratio had changed to 8:1. This evidence from learning confirms the importance of structures, although clearly the reasons for their power in learning are not the same as for use.

When it comes to using structures, Gardner relates studies of mathematicians' introspections about finding successful solutions to problems (my emphasis):

> "Sometimes the intuition comes through first, and then one must actually make efforts to work through the details of the solution; at other times, *the careful execution of the steps actually suggests the solution*; less frequently, intuition and discipline arrive at the same time or work in concert."

These processes and styles of activity correspond very strongly to those which compose MOR.

MOR is an interplay of structure, speculation and discovery. Structural knowledge guides our decision to speculate and our choice of speculation to attempt. Further structural knowledge guides us in discovery, as when we use wavefront markers to align the portions of expressions to be matched, and select specific methods to effect proof. Discovery of values which confirm the appropriacy of a speculation affirms our selection of a suitable structure for a problem.

The nature of the co-operation of human and machine is still heavily human-controlled in the systems described in this research. The human recognises structure and gives it to the machine. Eventually more of the recognition and its representation may be automated too.

Rôle of Heuristics

Schoenfeld sets heuristics in a larger context which explains more about useful rôles for them, and hence suggests different uses of MOR [Schoenfeld 85]. An admirer of Polya's work, he warns that attempts to teach his heuristic approach have not succeeded in the wider sense. Although Polya's descriptions strike a chord with many successful mathematicians, teaching them to students only seems to improve their performance on problems similar to the ones they learnt on, and couched in language which is suggestive of an appropriate technique.

He attributes this to the broadness of Polya's heuristics and difficulties in controlling their use. A typical failing is that people often set out along the wrong track altogether, attempting the wrong technique for the problem. They stumble in the initial stage of translating from the problem to mathematical model. By constructing an ill-chosen model, they are led to unsuitable techniques.

Schoenfeld's analysis of the knowledge and behaviour necessary for characterising mathematical problem-solving performance has four components:

- **Resources.** Mathematical knowledge possessed by the individual.
- **Heuristics.** Strategies and techniques for making progress on unfamiliar or nonstandard problem.
- **Control.** Global decisions regarding the selection and implementation of resources and strategies.
- **Belief Systems.** One's "mathematical world view" about self, the environment, the topic and mathematics.

In terms of this analysis, the Edinburgh mathematical reasoning group's work has concentrated on the heuristic level. MOR relies on working within a particular heuristic, using its structure and supplying detailed information on the use of resources. Although we would like to use it to select *between* heuristics, by indicating whether or not they apply, that is difficult when a reason for failure may be just lack of information at that point.

Given the great need for structure in directing MOR, I believe it will be a long time before major progress is made on using MOR at what Schoenfeld describes as the control level, since this is a level at which a structure is selected.

11.2 Further Work

11.2.1 Immediate Improvements

I view my implementation as a pilot study, an experiment with MOR. Simplifying assumptions made to explore the overall idea need to be revised and consolidated into robust long-term versions. With this in mind, there are a number of implementation and interface details which I would improve if MOR were to become a regular feature. Some of these have already been noted in section 11.1, especially improved handling of types.

Tail-Recursive Optimisation

The wave method I built to incorporate meta-variables assumes that there is a single wavefront in the conclusion. This avoids the problem of working out which wavefront in the target expression corresponds to which one in the rule. In all the problems attempted so far, this assumption is valid. Accommodating more wavefronts should be a minor task, especially now that CLAM has more machinery for handling complex wavefront patterns.

My supermethod for tail-recursive optimisation uses a submethod called *ripple-over* to perform a longitudinal ripple followed by a transverse one. The former provides evidence the latter uses to identify the meta-variable. This is purely an economy measure. For all the problems in my repertoire, it is exactly what I need. Straightforward variations on this pattern will extend it to cover most of the other transformation covered by Huet& Lang's templates (chapter 8). It will also suffice until using meta-variables is extended to permit their identification through progressive unifications (described at the end of 11.2.2), or control of rippling is made more subtle, as described next. In fact, the system is capable of finding the informative step case part of all these proofs by the iterative deepening planner, just using freestanding methods.

In the justification branch of these proofs, where the generalised formula is used to deduce the original and assure their equivalence, the accumulator inserted must be given a base value. The situation I have in mind is the proof branch starting as:

$$\forall x' \forall \vec{z'} \forall a \exists y'.y' = g(f(x', \vec{z'}), a)$$
$$\vdash \forall x \forall \vec{z} \exists y.y = f(x, \vec{z})$$

The proof proceeds by instantiating x' to x, and z' to z, but is then faced with a decision for the value of a, the accumulator:

$$\forall a \exists y'.y' = g(f(x, \vec{z}), a)$$
$$\vdash \exists y.y = f(x, \vec{z})$$

A value A must be chosen for a in such a way that the resulting value for y', can be used for y. For this, we will need

$$g(f(x, \vec{z}), A) = f(x, \vec{z})$$

and since this must be true for any values of x and $\vec{z}$, we really need A such that more generally:

$$\forall v.g(v, A) = v$$

Currently, MOR uses knowledge of available lemmas to select a suitable value for A, as described in chapter 7. A reduction rule or base case is sought which suggests a value immediately. In general some computation and proof might be required to find a suitable value, requiring more extensive MOR.

Enhanced Proof Structure Information

Newer versions of CLAM record more detailed information about the direction of wavefronts and the positions of potential accumulators or sinks. They have better routines for joining and splitting multiple wavefronts. Indeed since I froze a version of CLAM to work on, the entire characterisation of wave rules has been

refined and extended considerably. An obvious improvement on what I have done would be to make use of this, and deal with conditional wave rules and the presence of more than one wave (as in fibonacci, for example).

This would have implications for the choice of accumulator. Currently a universally quantified variable is inserted deliberately to fulfill the function of being an accumulator, and identified by name. In general, there might be other universally quantified variables present which could perform that function. The associated problems would be detecting that this might be possible and the selection of which universal variable to use. This does not mean that tail-recursive optimisation is only ever needed for univariate functions. The other arguments could be of the wrong type, they could be too bound up in the current definition to admit another usage, or the available wave rules might not permit their argument positions to be used.

Matching Abstracted Terms

As described in Chapter 4, in order to constrain the unification process and ensure that the control information available from wavefronts is used, I apply the unification algorithm on progressively larger terms, so that only matching is used in the successful solution branch. Each of these larger terms includes the previous ones.

This kind of idea is certainly needed, but could also have been achieved by λ-abstracting the entities to be unified and then unifying those abstractions. It could be awkward to do in any non-trivial case, as I shall now show. Given a conclusion which has already rippled longitudinally:

$$\vdash \forall \vec{z} \forall a \exists y.y = F(\boxed{c'(\underline{f(x,\vec{z})}, x, \vec{z})}^{\uparrow}, a)$$

and a wave rule:

$$g(\boxed{c'(\underline{U}, V, \vec{W})}^{\uparrow}, A) \rightarrow g(U, \boxed{c''(V, \vec{W}, \underline{A})}^{\downarrow})$$

Instead of matching progressively:

1. $f(x, \vec{z})$ and U
2. $c'(f(x, \vec{z}), x, \vec{z})$ and $c'(f(x, \vec{z}), V, \vec{W})$ (U having been instantiated)
3. a and A
4. $F(c'(f(x, \vec{z}), x, \vec{z}), a)$ and $g(c'(f(x, \vec{z}), V, \vec{W}), a)$

we would match

1. $f(x, \vec{z})$ and U
2. $\lambda t.c'(t, x, \vec{z})$ and $\lambda t.c'(t, V, W)$ - abstracting over the hole;
3. a and A
4. $\lambda v.F(v, a)$ and $\lambda v.g(v, a)$ - abstracting over the wavefront.

This would be a partial improvement, but we would still need the filtering, to stop a from being built into the identity of F. More advanced wave rule analysis would be needed to see the possibility of λ-abstraction over a as well, as a is universally quantified. Then there would be no need for solution-filtering to impose temporal scoping restrictions. In general, identifying the form of the abstracted function is a non-trivial task in its own right.

Control of Unification

As I wrote in chapter 2, Miller's algorithm has some desirable properties, and it would be interesting to see whether it could be used for all the tail-recursive synthesis work, or whether some adaptation would be necessary. Miller notes [Miller 90] that Huet's algorithm would also serve for his task " if ... [it] ... is modified to handle a mixed quantifier prefix and to solve those flexible-flexible equations". So it may well be that this problem is better approached from the other direction, of modifying Huet's algorithm to achieve those desirable properties, without making all the restrictions that Miller does.

Other Recursive Types

The current system only works on lists and natural numbers. Extending it to handle integers should be straightforward, and less similar types such as trees, not very much harder. Much of the difficulty in proofs about other structural types is in enhancing the type-guessing routines.

11.2.2 Next Steps for *MOR*

Tail-Recursive Optimisation

As I noted in subsection 11.1.2, the system can produce alternative programs depending on the lemmas available to the system. Currently, it just picks the first one that works. It would be entirely feasible to select amongst these and choose "the best" of them, perhaps satisfying some other desirable property, provided that we had a means of making such decisions.

What I have in mind is that at the moment, the system accepts any transverse ripple which permits the induction hypothesis to be used without surrounding it by function applications, i.e. without creating a stack. This does not discriminate between different uses of the accumulator. Poor choices here may mean that the stacking has just been moved onto the accumulator. Take the following versions of associativity of append (<>), which are transverse rules, but the second is more specialised:

$$\boxed{(\underline{X} <> Y)}^{\uparrow} <> Z \;:\Rightarrow\; X <> \boxed{(Y <> \underline{Z})}^{\downarrow} \tag{11.1}$$

$$\boxed{(\underline{X} <> (W :: nil))}^{\uparrow} <> Z \;:\Rightarrow\; X <> \boxed{(W :: \underline{Z})}^{\downarrow} \tag{11.2}$$

Each of these rules might apply to the conclusion:

$$\vdash \forall a \exists y.\, y = \boxed{(\underline{rev(t)} <> (h :: nil))}^{\uparrow} <> \lfloor a \rfloor$$

Both would result in a program fragment which was officially tail-recursive. However, the application of (11.2) would not build up a stack of functions to be

executed, since :: is a constructor symbol. This is preferable, although either version would be tail-recursive.

Ideally, using (11.1), we would look for another wave rule to ripple the wave-front further in and produce the constructor function, so that after reaching

$$\vdash \forall a \exists y.\, y = rev(t) <> \boxed{(\lfloor (h :: nil) <> \underline{a} \rfloor)}^{\downarrow}$$

the accumulating term would be simplified. The current version of CLAM now does this automatically. Such simplification would be necessary even if the dominant functor were a constructor, since subterms might not be, and they might yield to an appropriate sequence of ripples and simplifications. As there could be alternative simplifications, what is "maximal" would have to be established. Space would be saved, since each subsequent function call's description would be compacted.

Chapter 7 has described structural relationships between transformation templates and a rippling/fold-unfold approach to synthesising tail-recursive programs. The MOR system should be extended to cover the other patterns which it does not presently address. This would necessitate extending the supermethod which controls the proof structure.

More radically, we could try to get MOR to find *any* proof which eventually satisfied the proof-theoretic criterion for tail-recursiveness. This could require control of more complex conditionals not built into induction schemes, more elaborate rippling patterns, and perhaps use of properties such as commutativity, which we have avoided.

A different tack would be to take the computation induction proofs used to ensure equivalence in Huet and Lang's paper would admit MOR themselves. It might be interesting to explore the characterisation of **those** computation induction proofs within CLAM as a general template finding system. This would experiment with higher-order transformation templates in general.

Extending Structural Guidance for Tail Recursion

Péter's classic study of recursive functions [Péter 67] may provide further ideas for proof structures. She shows how certain classes of recursively defined function admit translations into equivalent primitive recursive forms. Some of her proofs construct the transformed function from the original to a new function with the desired form. Others, however, only use proof by contradiction to prove that there must be such a form, using majorisation, so they will not readily lead to the construction of a structure.

Other Program Optimisation Techniques

Other techniques exist which produce various kinds of economies in programs, such as tupling [Feather 79]. Tupling has been described within the rippling paradigm, and characterised through proof structures. It would be interesting to take other such techniques and do likewise.

If this only led to their emulation, then we would still have extended our system. If it led to a deeper understanding of these heuristics and perhaps their extension, a significant gain would have been made. Some such techniques might admit elements of MOR.

A result of characterising different types of optimisation within the same framework would be that we would have to work out how to select or combine when more than one was available. A goal of description within a common framework would be the ability to analyse interactions. The proofs-as-programs paradigm should provide a good basis for studying this problem, simultaneously encompassing functional and proof-theoretic properties.

Other Current Work on Existentially Quantified Variables, Choosing Inductions and Synthesis

The work reported here should be integrated with work on identifying existentially quantified variables and choosing inductions, currently underway in the

mathematical reasoning group in the Department of Artificial Intelligence, University of Edinburgh. In this, existentially quantified variables are viewed as having potential wave structures within them since they may be dependent on the induction variable. Here then, there is ample scope for MOR to identify such variables progressively, as the proof demands. This may well happen at the same time as trying to work out which induction to perform, as will be seen in the example below.

Often these existentially quantified variables are used for specifications described in other ways than the style used in this work. A common approach puts conditions on the existentially quantified variable, in the form of surrounding predicates. By working through a sample proof, we can see how MOR could contribute, and what the problems would be. I shall use capital letters to denote meta-variables, as usual.

An example is the prime factorisation theorem, where we prove that any positive integer x can be expressed as the product of a list of prime numbers, y:

$$\forall x \exists y. \mathit{product}(y) = x \tag{11.3}$$

MOR ties down the value of y to make the proof work, and finds a suitable induction.

First, an induction, since there's nothing else. We leave variable the exact induction scheme. Most guidance should be found in its step case or cases of the form:

$$\exists y'.\ \mathit{product}(y') = x \vdash \exists y.\ \mathit{product}(y) = \boxed{C(\underline{x})}^{\uparrow} \tag{11.4}$$

C is representing some arbitrary step case. We hope to select an induction by MOR identifying a possible value. Note that for an induction with multiple step cases, a more elaborate analysis would be needed, either of all the ways this could be satisfied, or by accepting an induction scheme if only one of its step cases was suggested.

Now we speculate about y. Note that although the meta-variable used for y looks as if it will be a skolem function of x, that is misleading. Thinking about

the ensuing induction proof, we expect y to be a function of y', and dependence on x to be introduced that way, through y''s dependence on x. So further $\mathcal{MOR}$ would suggest that y should be regarded as $Y(y')$, and that assumed for the existential witness:

$$product(\boxed{Y(\underline{y'})}^{\uparrow}) = \boxed{C(\underline{x})}^{\uparrow} \tag{11.5}$$

This also ties the wavefronts together in an intuitively satisfying way.

In accordance with our knowledge of induction proofs, we expect to ripple now. Let us assume the following wave rules:

$$product(\boxed{Hd :: \underline{Tl}}^{\uparrow}) \Rightarrow \boxed{Hd \times \underline{product(Tl)}}^{\uparrow}$$
$$\boxed{U \times \underline{V}}^{\uparrow} = \boxed{U \times \underline{W}}^{\uparrow} \Rightarrow V = W$$

In principle we would require a heuristic to decide which wavefront to work on, left or right. We could try to decide which was most likely to be informative, either by virtue of being most instantiated, or providing most information on being attempted. CLAM is lacking in such heuristics at the moment. We could probably decide that the left hand side of (11.5) is most instantiated.

In this case only the first of the two wave rules above could be applied. Using the definition of *product*, and instantiating $Y(y')$ as $Y_1(y') :: Y_2(y')$, this can be rippled:

$$\boxed{Y_1(y') \times \underline{product(Y_2(y'))}}^{\uparrow} = \boxed{C(\underline{x})}^{\uparrow} \tag{11.6}$$

Now a simple $\mathcal{MOR}$ would identify C by seeing that the wavefronts can be rippled past the equality, the second wave rule above, if $C(x)$ is $Y_1(y') \times C_2(x)$. We get:

$$product(Y_2(y')) = C_2(x) \tag{11.7}$$

and since we would now expect strong fertilisation to apply, it can instantiate Y_2 and C_2 to the identity function. It should not be too difficult to work out that $Y_1(y')$ is just the new free variable introduced for the induction, and has

no actual dependence on y'. This should be enough to identify the induction scheme.

We would have to do some work to identify Y more generally, combining it with a base case value.

This example is artificially simple because so few wave rules are present, and there are no case-splits, but it shows the approach has potential.

Conjecturing Lemmas

Within strong proof structures, it could happen that we would have a proof which almost succeeded, but lacked some crucial lemma. If a general form were known for such a lemma, because of the rôle it was expected to perform, it might be possible to conjecture an appropriate lemma, and allow MOR proof to fill in the details.

For example, we might use this for dynamic ripple formation - the ability to combine the use of a collection of lemmas so that their totality constituted a ripple. This would allow us to build ripples using commutativity, for example, which are not available to us now, unless we work out all the possible permutations.

Progressive Instantiation of Meta-Variables

Currently, instantiation of meta-variables is a top-down operation. We unify a variable X and a constant c, and the result is c. This is inherent in the algorithm.

Of course it is up to us how and to what we apply it. In fact there are occasions when we would wish to be more flexible. Choosing to unify the whole of X with c denies the possibility that X could be composite, $X'(X'')$, say. In which case we might have reason to explore the possibility that just X'' unifies with c. This would leave X' to be identified subsequently.

Of course there could be an ascending chain of $X'(X''(\ldots X^{\cdots})$; we could always postulate the existence of a composite variable like this and rely on the

possibility of outer functional meta-variables being projections to reproduce the straightforward case which has been sufficient in all my examples so far.

This could be useful in the newer versions of CLAM if we were trying to use a sequence of ripples to achieve the transfer of the wavefront as required, and the functional meta-variable corresponded to a just such a composite function. However there is an obstacle to implementing this now, in that we would need to know the types of all the meta-variables X, X' and X'' at the outset for the unification algorithm. We may be able to deduce X's type, but that would be all we would know about X' and X'', i.e. there would be a "type gap" between them. David Pym's algorithm should make it possible to find these unknown types.

11.2.3 Extending Proof Structure for Conditionals

In the constructive logic framework, tail-recursion is determined by the way we use the induction hypothesis. The witness for this hypothesis is the construction corresponding to the recursive call in the function definition. If we are able to use it (or one of its subterms) alone as the justification of the goal, then no further work is required after the recursive call, and the function is tail-recursive. If instead we have to embed it in some function, that function application enters into the construction, which is no longer tail-recursive. In my definition of tail-recursion, I mentioned the possibility of conditionals. The function surrounding the recursive call could just be a condition. That is a complication I do not address in these proof structures, and it is an avenue for further work.

Generalisation

Signs of success or failure in a particular proof are still only partially understood, and would benefit from more research. To be able to control the proof of the distributivity of $\times$ over $+$ when all variables are x, this would be needed.

Advanced reasoning about the nature of equality is an essential element of this. The analysis I provided in chapter 9 on failure and success in induction is

only a rough guide to far more elaborate proof-theory. Such theory should be adaptable to provide us with more precise heuristics. Problems with it are that it is usually directed at number theory only, and that the induction schemes it assumes are far stronger than any we have access to.

The proof theoretic notion of "if this were provable, it would be by steps S_1/would have been by steps S_2" is something we make little use of in our work, yet there are such results [Sieg 90]. With adaptation to the theories we use, they might help us build heuristics suggesting how many inductions should be required for a successful proof attempt, or how many sinks.

11.2.4 Other Applications of *MOR*

Other Types of Generalisation

Within the category of generalising to assist induction, it is possible to generalise in other ways than I have done so far. I described these in chapter 9.

I have made attempts to generalise constants to act as induction variables. This is a standard technique mathematicians use, and is an obvious next step, using techniques similar to those I have already described for variables. Van der Waerden's proof of Baudet's conjecture [der Waerden 71] does this, starting with a conjecture about splitting the positive integers into two classes, and eventually proving its equivalent for any number of classes.

These kinds of generalisation are simple cases of the overarching result that proving any proposition over all the members of a set covers all its subsets or indeed any member. Many generalisations are in this category. Some of these should be quite accessible to *MOR*.

Complementary to these generalisations is the technique of weakening hypotheses. Van der Waerden's proof uses this too, by proving a result assuming only a finite subset instead of the infinite set of positive integers.

Clearly there are many more forms of generalisation, at least some of which should be accessible to *MOR*. In all of them, as with the examples I have studied

so far, an analysis of what generalisation achieves should drive this, if one is available. Otherwise it is very hard to know when to initiate the generalisation.

A theoretical source of information may be found from cut elimination results. These show whether or not the cut rule of inference is needed for a given theory expressed in a given logic. This is relevant because the cut rule of inference is used to insert the generalised version of the theorem into the proof. If the cut rule is not necessary, generalisation is not necessary either. If the cut rule is necessary, then the point at which the proof of the cut elimination theorem breaks down may be a pointer to the types of generalisation which may be needed.

Cut elimination theorems are constructive, working through all the circumstances in which a cut might be used, and showing how in each case it could be replaced by some other cut-free proof fragment. Hence the nature of a failure indicates certain key features about the theory, which are liable to need cuts, and hence generalisation. These might indicate some kinds of generalisation which might be needed, and indicate enough about their form to suggest how MOR could be used.

Even if they are not necessary for proof, certain kinds of generalisation could still be necessary to achieve special effects on the extract term as with the tail-recursion proofs.

Learning from Failure

Learning from failure forms part of the guidance in instigating a MOR attempt, and in suggesting a speculation. Boyer & Moore also use such techniques to recognise the need to use induction in their theorem prover.

More widely, though, information about *how* failure has occurred is rarely used to suggest likely next steps. This should be a promising avenue for MOR, because often failures can be quite localised, and therefore amenable to patching. MOR is an obvious tool to assist with such patching.

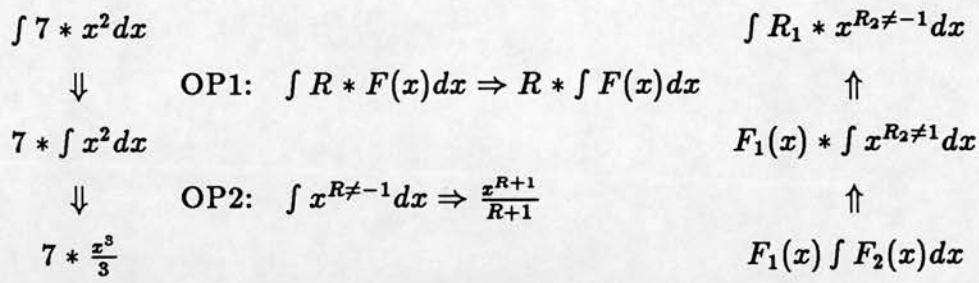

Figure 11–1: Example of LEX2's Constraint Back-Propagation

Learning

A candidate for the use of higher-order meta-variables, but not particularly MOR, is the *construction* of patterns for problem solving. In this thesis the patterns have been supplied, and the system has used them to find solutions. An alternative way of using meta-variables is for discovering patterns.

LEX2 [Mitchell *et al* 83] uses constraint back-propagation to learn kinds of *tactics* which are sequences of operators, and preconditions for the tactics which are the combined preconditions of the operators. A small example is in figure 11.2.4 The solution to the integration problem (left-hand column) along with the operators used to create it (centre) are givens. Their tactic is progressively found in the right-hand column.

The letters used have special significance, denoting the types of objects they may represent in the system's elaborate generalisation hierarchy in which

F may denote any first-order function

R may denote any rational number

In [Mitchell *et al* 83], the authors permit multiple uses of the same type-recording letter to stand for different instances in the right-hand column, but not in the rules. I have differentiated different instances for clarity, and convenience.

The task is to establish the properties of the initial expression which made the chain of rewritings successful. This is worked out backwards, looking at each operator, and seeing what the expression must have been like at the stage when

the operator applied, in order for it to be applicable. We know from the form of the final expression in the actual solution that the end point is the product of two terms.

The preconditions of $OP2$ are embodied in its left-hand side - an expression of the form $\int x^R dx$ where R is not -1. $F_2(x)$ must be x^{R_2}.

Continuing to work backwards, this has to be all or part of the expression resulting from applying $OP1$, so the right-hand side of $OP1$ must fit it. This forces $F_1(x)$ to be a rational number, R_1, say.

The preconditions of the tactic are now a (sub)term of the form

$$\int R_1 * x^{R_2 \neq -1} dx$$

LEX2 avoids using higher-order meta-variables by using its generalisation hierarchy of types of function. Instead of more information restricting identity by unifying, it moves down the hierarchy to a more specific type.

An alternative would be to

- skip the separate type hierarchy,
- use meta-variables for the functions and expressions, and
- use unification to accumulate the preconditions.

Robin Boswell, an earlier student in the Mathematical Reasoning Group, started to explore this topic, but did not pursue it far [Boswell 89].

Tutoring

If we believe that the $\mathcal{MOR}$ approach has improved our ability to emulate human thought processes this brings closer the possibility of using $\mathcal{MOR}$, and specifically the CLAM system, as part of a tutoring system, in which the computer could track the process the human was/should be attempting.

Tuition could concentrate on planning, heuristics or proof.

Improving Interactive/Co-operative Proof Planning

In an interactive system it should be possible to let the human insert meta-variables standing for object-level entities as they wish. The computer system could suggest instantiations or directions for proof. The human could accept or override these, proceeding with the proof while computer noticed any constraints and acted accordingly.

Conceivably, the human could indicate a desired goal state using meta-variables and leave it to computer to reach it.

Search Control

The tail-recursive $\mathcal{MOR}$ system I described only ever used one meta-variable at once, within an overall proof structure. For the work on generalisation with multiple meta-variables and wavefronts, choosing what to work on first became more problematic. I described some simple heuristics in chapter 9.

In more general cases, with multiple subgoals to decide amongst as well, an extended search control heuristic might be needed, paying attention to a number of other measures.

We might choose to develop the most instantiated subgoals or wavefronts. With plenty of information, there should be least search available and fastest rejection of failure branches. In the case of choosing subgoals, this might be refined if we expected to apply a particular method. In that case, we might only be interested in the degree of instantiation of the subterm on which the method would act.

Alternatively, we might look for cases involving meta-variables where the options were fewest, so the search space branched least, and we could hope to find a successful branch easiest from a small number.

Another possibility is to try to make choices which are "most informative" in reducing the number of meta-variables.

Selecting a heuristic to achieve this is a research topic in itself. It is a task equivalent to that faced by a meta-interpreter. For some of these options there is a considerable overhead even in discovering the information in order to make a decision.

Middle-Out Planning

The *MOR* in this work is purely at the object level. For greater flexibility and better human emulation, we should also attempt middle-out planning. The need for this becomes obvious just from considering the induction method. This method produces a variable number of base and step cases, depending on the induction scheme chosen. That is not a problem when we have chosen the induction scheme, since that determines the number and type of subgoals. If the induction scheme is what we want *MOR* to suggest, the planner must be able to accept for its list of output sequents a variable list of indeterminate length, with at most the assumption of one step case, whose structure remains to be fixed.

A harder example of middle-out planning is the use of a case-split. The need for one is usually only discovered at a deeper branch of the proof. This necessitates going back up the proof tree, choosing the best point for the insertion of the case-split, and then re-doing the resulting branches, possibly using the previous proof attempt.

Meta-Variable Types

As I noted in section 11.1 recent work by David Pym [Pym 90] extends Huet's algorithm to permit variables of dependent types to have the dependent parts of those types variable and then unify the objects' types as well as the objects. Implementing this would be a useful extension for two reasons. Firstly, we would be using a unification system better suited to the underlying logic. Secondly, our reasoning would be more thoroughly middle-out, we would not always need to commit ourselves in advance to the type of a meta-variable object, if it could be

inferred as the identity of the object was inferred, from the proof requirements. As yet, Pym's algorithm could not infer the types of all the kinds of meta-variables used in *MOR*. In practice, this change would not affect the tasks I have used here, but would make the whole technique's implementation more principled.

11.3 General Conclusions

MOR at the meta-level incorporating higher-order unification has been studied as a tool for guiding inductive theorem proving. It has been shown to be useful in two areas

- Tail-recursive synthesis. By using the proofs-as-programs principle to characterise proof structures which correspond to tail-recursive programs, we can synthesise programs from specifications. *MOR* deduces the identity of functions required for the new definition using the proof-planning methodology for control.

- Generalisation. *MOR* offers a principled and unified approach to a number of classes of generalisation previously treated separately.

Although higher-order unification can be hard to control, it is manageable. More control can be gained over it by knowledge about the type of unification required for a particular problem. Reduction to matching can be achieved through an understanding of proof structure.

In both of the areas for which *MOR* has been explored, the key to success has been the existence of a detailed analysis of the structure of successful proofs. Such structural knowledge will inevitably be needed in general, as there is no way that meta-variables can be controlled if they are not introduced for a purpose. Structural analyses of proofs yield information which can be used to

- guide the decision to attempt *MOR*,

- indicate how meta-variables should be introduced,
- control the proof steps which suggest the values of the meta-variables,
- restrict the unification and instantiation process.

The *MOR* technique is a considerable enhancement to our ability to describe plans for theorem-proving. This kind of flexible planning seems to correspond well with the behaviour of human mathematicians.

More broadly, *MOR* can be seen to be a valuable tool with great potential for assisting theorem-proving in a variety of problems and contexts.

Bibliography

[Aubin 76] R. Aubin. *Mechanizing Structural Induction.* Unpublished PhD thesis, University of Edinburgh, 1976.

[Bauer & Wössner 82] F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[Boswell 89] R.A. Boswell. Analytic goal regression: problems, solutions and enhancements. In Y. Kodratoff and A. Hutchinson, editors, *Machine and Human Learning*, pages 9–23, Michael Horwood Ltd, 1989.

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Boyer & Moore 88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[Bundy & Welham 81] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. *Artificial Intelligence*, 16(2):189–212, 1981. Also available from Edinburgh as DAI Research Paper 121.

[Bundy 83] A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983. Second Edition. Earlier version available from Edinburgh as Occasional Paper 24.

[Bundy *et al* 88] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. *Experiments with Proof Plans for Induction.* Research Paper 413, Dept. of Artificial Intelligence, Edinburgh, 1988. To appear in JAR.

[Bundy *et al* 90a] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. *The Oyster-Clam system.* Research Paper 507, Dept. of Artificial Intelligence, Edinburgh, 1990. Appeared in the proceedings of CADE-10.

[Bundy *et al* 90b] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146, Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al* 91a] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. *Rippling: A Heuristic for Guiding Inductive Proofs.* Research Paper 567, Dept. of Artificial Intelligence, Edinburgh, 1991. Submitted to Artificial Intelligence.

[Bundy *et al* 91b] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324,

1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Burstall & Darlington 77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[Castaing 85] J. Castaing. How to facilitate the proof of theorems by using the induction-matching, and by generalisation. In Aravind Joshi, editor, *Proceedings of the Ninth IJCAI*, pages 1208–1213, IJCAI, 1985.

[Cohn 79] A.J. Cohn. *Machine Assisted Proofs of Recursion Implementation.* Unpublished PhD thesis, University of Edinburgh, 1979. Available as CS tech report CST-6-79.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Cooper 66] D. C. Cooper. The equivalence of certain computations. *Computer Journal*, 9:45–52, 1966.

[Curry & Feys 58] H.B. Curry and R. Feys. *Combinatory Logic.* Volume 1, North-Holland, 1958.

[Darlington 72] J. Darlington. *A Semantic Approach to Automatic Program Improvement.* Unpublished PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1972.

[Darlington 81] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46, August 1981.

[Davis & Putnam 60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.

[der Waerden 71] Van der Waerden. How the proof of Baudet's conjecture was found. In L. Mirsky, editor, *Papers presented to Richard Rado on the occasion of his sixty-fifth birthday*, pages 252–260, Academic Press, London-New York, 1971.

[Desimone 89] R.V. Desimone. *Learning Control Knowledge within an Explanation-Based Learning Framework.* Unpublished PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[Ernst & Newell 69] G. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, 1969.

[Feather 79] M.S. Feather. *A System for Developing Programs by Transformation.* Unpublished PhD thesis, University of Edinburgh, 1979.

[Gardner 84] H. Gardner. *Frames of Mind.* William Heinemann Ltd., 1984.

[Gelernter 63] H. Gelernter. Realization of a geometry theorem-proving machine. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 134–52, McGraw Hill, 1963.

[Gilmore 60] P.C. Gilmore. A proof method for quantificational theory. *IBM J Res. Dev.*, 4:28–35, 1960.

[Giunchiglia & Walsh 89] F. Giunchiglia and T. Walsh. Theorem Proving with Definitions. In *Proceedings of AISB 89*, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1989. Also available from Edinburgh as DAI Research Paper No 429.

[Giunchiglia & Walsh 91] F. Giunchiglia and T. Walsh. Using abstractions. In *Proceedings of AISB-91*, Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1991.

[Gordon 88] M. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, page , Kluwer, 1988.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[Hannan & Miller 88] J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium*, pages 942–59, MIT Press, 1988.

[Harper *et al* 87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. of the Second Symposium on Logic in Computer Science*, 1987.

[Harris & Khoshnevisan 88] P.G. Harris and H. Khoshnevisan. *On the transformation of linear functions: a new approah to recursion removal.* Technical Report, Department of

Computing, Imperial College of Science and Technology, University of London, January 1988.

[Hill & Lloyd 88] P. Hill and J.W. Lloyd. Analysis of meta-programs. In J. Lloyd, editor, *Proceedings of the Meta '88 Workshop on meta-programming in Logic Programming*, Meta '88, June 1988. Longer version available as Technical Report CS-88-08, Department of Computer Science, University of Bristol.

[Horn 88] C. Horn. *The Nurprl Proof Development System.* Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[Howard 80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, 1980.

[Huet & Lang 78] G. Huet and B. Lang. Proving and applying program transformation expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

[Huet 75] G. Huet. A unification algorithm for lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hummel 87] B. Hummel. *An Investigation of Formula Generalisation Heuristics for Induction Proofs.* Interner Bericht 6/87, Universitaet Karlsruhe, June 1987.

[Hummel 90] B. Hummel. *Generation of induction axioms and generalisation.* Unpublished PhD thesis, Universität Karlsruhe, 1990.

[Jensen & Pietrzykowski 76] D.C. Jensen and T. Pietrzykowski. Mechanizing ω-order type theory through unification. *Theoretical Computer Science*, 3:123–171, 1976.

[Knight 89] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1), 1989.

[Lenat 82] D.B. Lenat. AM: an artificial intelligence approach to discovery in mathematics as heuristic search. In *Knowledge-based systems in artificial intelligence*, McGraw Hill, 1982. Also available from Stanford as TechReport AIM 286.

[Manna & Waldinger 74] Z. Manna and R. Waldinger. *Knowledge and Reasoning in Program Synthesis.* Technical Note 98, SRI International, Menlo Park, November 1974.

[McCune 89] W. McCune. *The Otter User's Guide.* , To be published as an Argonne National Laboratory Technical Report, 1989.

[Miller & Nadathur 87] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 1987.

[Miller & Nadathur 88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*, MIT Press, 1988.

[Miller 90] D. Miller. *A Logic Programming Language with Lambda Abstraction, Function Variables and Simple Unification.* Technical Report MS-CIS-90-54,

Department of Computer and Information Science, University of Pennsylvania, 1990. To appear in *Extensions of Logic Programming*, edited by P. Schröder-Heister, Lecture Notes in Artificial Intelligence, Springer-Verlag.

[Mitchell *et al* 83] T.M. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: acquiring and modifying problem-solving heuristics. In R.S. Michalski, J.F. Carbonell, and T.M. Mitchell, editors, *Machine Learning*, pages 163–190, Tioga Press, 1983.

[Partsch 90] H Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Paterson & Hewitt 70] M.S. Paterson and C.E. Hewitt. Comparative schematology. record of the project mac conference on concurrent systems and parallel computation, woods hole, mass. *ACM*, 119–127, 1970.

[Paulson 86] L. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[Péter 67] R. Péter. *Recursive Functions.* Academic Press, 1967. Translated by Istvan Foeldes.

[Piaget 52] J. Piaget. *The Child's Conception of Number.* Routledge & Kegan Paul, 1952.

[Plummer 85] D. Plummer. *Gazing: Using the Structures of the theory in theorem proving.* Working Paper 180, Dept. of Artificial Intelligence, Edinburgh, May 1985.

[Polya 45] G. Polya. *How to solve it.* Princeton University Press, 1945.

[Pym 90] D.J. Pym. *Proofs, search and computation in general logic.* Unpublished PhD thesis, University of Edinburgh, 1990. Available as LFCS report ECS-LFCS-90-125.

[Robinson 79] J.A. Robinson. *Logic: Form and function. The mechanization of deductive reasoning.* Edinburgh University Press, 1979.

[Schoenfeld 85] A.H. Schoenfeld. *Mathematical Problem Solving.* Academic Press, 1985.

[Sieg 90] W. Sieg. Herbrand analyses. *Archive for Mathematical Logic*, 30, 1990.

[Skemp 71] R.R Skemp. *The Psychology of Learning Mathematics.* Penguin, 1971.

[Snyder & Gallier 89] W. Snyder and J. Gallier. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, 8(2):101–140, 1989.

[van Harmelen 89] F. van Harmelen. *The CLAM Proof Planner, User Manual and Programmer Manual.* Technical Paper TP-4, Dept. of Artificial Intelligence, Edinburgh, 1989.

[van Harmelen 89] F. van Harmelen. *On the Efficiency of Meta-level Inference.* Unpublished PhD thesis, University of Edinburgh, 1989.

[Wainer 89] S.S. Wainer. Programs from proofs. 1989. Seminar given at the Department of Artificial Intelligence, Edinburgh.

[Wallen 90] L.A. Wallen. *Automated Deduction in Non-Classical Logics.* MIT Press, London, 1990.

[Wiggins 90] G. A. Wiggins. The improvement of prolog program efficiency by compiling control: a proof-theoretic view. In *Proceedings of the Second International Workshop on Meta-programming in Logic*, Leuven, Belgium, April 1990. Also available from Edinburgh as DAI Research Paper No. 455.

[Zhaohui *et al* 90] L. Zhaohui, R. Pollack, and P. Taylor. *How to Use Lego.* To appear as LFCS tech. report, Department of Computer Science, University of Edinburgh, 1990.

Appendix A

Tail-Recursive Synthesis Methods using $\mathcal{MOR}$

A.1 Tail-Recursive Synthesis Strategy Method

```
method(tr_gen_strat(seq(BGeneralised,new[gen])
                    then [(induction(Scheme,Var:T)
                           then CasesTactics),
                          JustificationTac
                         ]),
       H==>G,
       [applicable_submethod(H==>G,tr_gen(Generalised),_,
                             [NewH==>G,H==>Generalised]),
        applicable((H==>Generalised),induction(Scheme,Var:T))
       ],
       [
                          % identification branch - induction

        scheme(Scheme,Var:T,H==>Generalised,BSeqs,SSeqs),

                          % identification branch - step case
```

```
maplist(SSeqs,SSeq:=> %SSeq2-
                        (Waves then Fert),
         (          % iterate wave over the step-case:
          applicable_submethod(SSeq,
                               ripple_over(Waves),
                               _,
                               [SSeq1]),
                     % and try a fertilize step:
          Fert=strong_fertilize(Hyp),
          applicable(SSeq1,Fert,_,[])
         ),
         StepTactics
        ),


                     % identification branch - base case
                     % use sym_eval for all base-cases:
maplist(BSeqs,
        BSeq:=> %BSeq1-
                (apply(sym_eval(SE))
                 then (repeat intro) then intro(Value)
                 then apply(tautology(Taut))),
        (applicable(BSeq,   sym_eval(SE),_,[SE_BSeq]),
         applicable_submethod(SE_BSeq,
                              subexistential(EVar:EType,
                                             Value),
                              _,
                              [E_BSeq]),
         applicable(E_BSeq,
                     tautology(Taut),
                     _,
                     WF_SubGoals)
```

```
        ),
        BaseTactics),
 append(BaseTactics, StepTactics, CasesTactics) ,

                    % justification branch
 applicable_submethod(NewH==>G,
                     tr_justification(JustificationTac),
                     _,
                     []),
 hou_beta_reduce(Generalised,BGeneralised)
],
[],
tr_gen_strat(seq(Generalised,new[gen])
             then [(induction(Scheme,Var:T)
                    then CasesTactics),
                   JustificationTac
                  ])
).
```

A.2 Wave Method

```
        % Higher order unification routine is interfaced by
        % making a copy with hou-style ground variable for
        % Prolog variables, and working on the copy. This
        % means constantly explicitly applying substitutions.

method(wave(Pos,[Rule,Dir]),
       H==>G,
       [wave_rules(WaveRules),
        long,
        wave_fronts(PureSequent,_,H==>G),
        hou_fresh_variables,
                                    % grounds vbles
                                    % and records types
        hou_set_up(H==>G,Gr_H==>Gr_G,[],SomeGroundList),
        matrix(Vars,Matrix,Gr_G),
                                    % get wavefronts
        wave_fronts(PureMatrix,WaveFronts,Matrix),
                                    % pick one
        member(WO-SWIs,WaveFronts),
                                    % find full hole paths
        maplist(SWIs,SWI:=>WI,append(SWI,WO,WI),WIs),
                                    % get rule in orig form
        member(RuleName:Rule,WaveRules),
                                    % get waverule
        wave_rule(RuleName,long(Dir),L:=>R),

%           Experimental fast rejection of irrelevant rules -
%           there is a function in L which is in G
```

```
        thereis{Function}:(exp_at(L,[0|_],Function),
                           exp_at(G,[0|_],Function)
                          ),

%       Use of prolog variables for meta-variables means that
%       we must do a lot of working typing things. The unifier
%       works on ground, labelled variables. For the sake of both
%       efficiency, and having all the information there when you
%       need it, I do all the type checking and instantiation
%       here, rather than each time I want to do a bit of
%       unification. I then work with the copies until the tactic
%       has succeeded, when I instantiate the real thing.

                                    % grounds vbles with
                                    % atoms, beta-reduces,
                                    % and records types
        hou_set_up(L:=>R,Gr_L:=>Gr_R,Rule,SomeGroundList,
                                          GroundList),
        wave_fronts(PureL,[LWaveOuterFront-SubLWaveInnerFronts],
                                                    Gr_L),
                                    % & its full hole paths
        maplist(SubLWaveInnerFronts,
                LI:=>LO,
                append(LI,LWaveOuterFront,LO),
                LWaveInnerFronts),
                                    % ho unify all holes
        maplist(LWaveInnerFronts,
                HI:=>Subst_for_Hole,
                                    % hole in rule
                (exp_at(PureL,HI,LH),
                                    % find hole in
                 member(WI,WIs),
```

```
                                % sequent and
            exp_at(PureMatrix,WI,GH),
                                % match
            hou_sub(LH,GH,[],Subst_for_Hole,_,_)
           ),
           HO_LHolesSubs
          ),
    hou_combine_subst_lists([],HO_LHolesSubs,HolesSubsts),
                                % and subst in L:=>R
    hou_apply_subst_set(Gr_L:=>Gr_R, HolesSubsts,
                                    Gr_L1:=>Gr_R1),
                                % and in H==>G
    hou_apply_subst_set(Gr_H==>Gr_G, HolesSubsts,
                                    Gr_H1==>Gr_G1),

% wave front     assumes just 1

    wave_fronts(Pure_Gr_L1,_,Gr_L1),
    exp_at(Pure_Gr_L1,LWaveOuterFront,LF),
    matrix(_,Matrix1,Gr_G1),
    wave_fronts(PureMatrix1,_,Matrix1),
    exp_at(PureMatrix1,WO,GF),
                                % actually match
    hou_sub(LF,GF,HolesSubsts,_,FrontSubst,_),
                                % & subst in L:=>R
    hou_apply_subst_set(Gr_L1:=>Gr_R1, FrontSubst,
                                      Gr_L2:=>Gr_R2),
                                % & in H==>G
    hou_apply_subst_set(Gr_H1==>Gr_G1, FrontSubst,
                                      Gr_H2==>Gr_G2),
```

```
% whole LHS
        wave_fronts(Pure_Gr_L2,_,Gr_L2),
        matrix(_,Matrix2,Gr_G2),
        wave_fronts(PureMatrix2,_,Matrix2),
                                    % must be strictly more
        append([_|_],Pos,W0),       % than the wavefront
        exp_at(PureMatrix2,Pos,Wave),
        hou_sub(Pure_Gr_L2,Wave,FrontSubst,FnSubst,WaveSubst,_),
                                    % need to trap any
                                    % substs which embed
                                    % a universal variable
                                    % in a goal meta-var
        setof(Univ,hou_universal_var(Gr_H2==>Gr_G2,Univ),Univs),
        forall {U\Univs}:(not (member(hou_subst(FS,GV),FnSubst),
                               exp_at(Gr_G2,_,GV),
                               exp_at(FS,_,U))),
        hou_commentary([
                        ['FnSubst:',FnSubst]
                       ])
      ],
      [
                                    % having operated on a
                                    % copy, fix real one
        hou_apply_subst_list_and_instantiate(WaveSubst,
                                             GroundList,H==>G),
        hou_apply_subst_list_and_instantiate(WaveSubst,
                                             GroundList,R),
        matrix(Vars,NonGroundMatrix,G),
        replace(Pos,R,NonGroundMatrix,NewMatrix),
        matrix(Vars,NewMatrix,NewG),
        hou_commentary([['H==>NewG:',H==>NewG]
                       ])
```

```
 ],
 [H==>NewG],
 wave(Pos,[Rule,Dir])
).
```

Appendix B

Examples of Tail-Recursive Synthesis Plans

For the first example, the whole run is shown, including loading all the rules etc. In subsequent examples, all the same information is loaded, and that part of the run has been omitted.

The $\mathcal{MOR}$ goal, the substitution, the generalised goal, and the plan produced at the end has been formatted to be more readable, and annotated.

B.1 Reverse

```
Script started on Wed Sep 25 10:20:16 1991
achtriochtan:iter_methods> testclam4.qui

Quintus Prolog Release 2.5.1 (Sun-4, SunOS 4.1)
Copyright (C) 1990, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

CLaM Proof Planner Version 1.4 (libraries only) (28/7/91 19:05)

CLaM Proof Planner (with HOU) Version 1.4 (9/9/91 15:28)
```

```
| ?- [['revload'].
[consulting /home/sin1/jane/oyster/thmlib/hou/iter_methods/revload.pl...]
 [consulting /home/sin1/jane/oyster/thmlib/hou/iter_methods/defs_and_waves.
loading def(app)...done                    /* append of
loading eqn(app1)...done                   /* two lists
loading eqn(app2)...done
 adding wave-record for app2...done
 adding recursive-record for app...done
loading def(rev)...done                    /* reverse of
loading eqn(rev1)...done                   /* a list
loading eqn(rev2)...done
 adding wave-record for rev2...done
 adding recursive-record for rev...done
loading def(plus)...done                   /* plus of
loading eqn(plus1)...done                  /* 2 numbers
loading eqn(plus2)...done
 adding wave-record for plus2...done
 adding recursive-record for plus...done
loading def(total)...done                  /* total of a
loading eqn(total1)...done                 /* list of numbers
loading eqn(total2)...done
 adding wave-record for total2...done
 adding recursive-record for total...done
loading def(length)...done                 /* length of
loading eqn(length1)...done                /* a list
loading eqn(length2)...done
 adding wave-record for length2...done
 adding recursive-record for length...done
loading scheme(twos)...done                /* 2-step induction
loading synth(half)...done                 /* integer half
loading def(half)...done
```

```
loading eqn(half1)...done
loading eqn(half2)...done
loading eqn(half3)...done
 adding wave-record for half3...done
 adding recursive-record for half...done
loading def(max)...done                  /* max of 2
loading eqn(max1)...done                 /* numbers
loading eqn(max2)...done
loading eqn(max3)...done
 adding wave-record for max3...done
 adding recursive-record for max...done
loading def(greatest)...done             /* greatest of a
loading eqn(greatest1)...done            /* list of numbers
loading eqn(greatest2)...done
 adding wave-record for greatest2...done
 adding recursive-record for greatest...done
loading def(times)...done                /* multiplication
loading eqn(times1)...done               /* of two numbers
loading eqn(times2)...done
 adding wave-record for times2...done
 adding recursive-record for times...done
loading def(sigma0)...done               /* summation of
loading eqn(sigma01)...done              /* the values of
loading eqn(sigma02)...done              /* a function
 adding wave-record for sigma02...done   /* from 0 to n
 adding recursive-record for sigma0...done
loading thm(assmax)...done               /* associativity
 adding wave-record for assmax...done    /* of max
 adding wave-record for assmax...done
 adding wave-record for assmax...done
 adding wave-record for assmax...done
loading thm(transsplus)...done           /* s(x)+y = x+s(y)
```

```
 adding wave-record for transsplus...done
 adding wave-record for transsplus...done
loading thm(assp)...done                    /* associativity
 adding wave-record for assp...done         /* of plus
 adding wave-record for assp...done
 adding wave-record for assp...done
 adding wave-record for assp...done
loading thm(assm)...done                    /* associativity
 adding wave-record for assm...done         /* of times
 adding wave-record for assm...done
 adding wave-record for assm...done
 adding wave-record for assm...done
loading thm(assapp)...done                  /* associativity
 adding wave-record for assapp...done       /* of append
 adding wave-record for assapp...done
 adding wave-record for assapp...done
 adding wave-record for assapp...done
loading thm(dist)...done                    /* distributivity
 adding wave-record for dist...done         /* of times over
 adding wave-record for dist...done         /* plus
 adding wave-record for dist...done
 adding wave-record for dist...done
loading thm(length)...done                  /* length step
 adding wave-record for length...done       /* case as theorem
loading thm(half)...done                    /* half step case
 adding wave-record for half...done         /* as theorem
 adding wave-record for half...done
loading thm(appsingle)...done               /* app(x,y::z)
 adding wave-record for appsingle...done/* =app(app(x,
 adding wave-record for appsingle...done/*  y::nil),z)
loading thm(plus2right)...done              /* x+s(y)=s(x+y)
 adding wave-record for plus2right...done
```

```
 adding wave-record for plus2right...done
loading thm(times2right)...done          /* times(x,s(y))
 adding wave-record for times2right...done/*=x+times(x,y)
loading thm(app1right)...done            /* app(x,nil)=x
 adding reduction-record for app1right...done
loading thm(plus1right)...done           /* x+0=x
 adding reduction-record for plus1right...done
loading thm(times1right)...done          /* times(x,0)=0
 adding reduction-record for times1right...done
 [defs_and_waves_load.pl consulted 11.783 sec 109,932 bytes]
loading thm(trrev)...done
[revload.pl consulted 11.850 sec 111,404 bytes]

yes
| ?- display.
trrev: [] incomplete autotactic(idtac)
==> x:pnat list=>y:pnat list#y=rev(x)in pnat list
by _


yes
| ?- dplan.

H==>NewG:
[v0:pnat,
 v1:pnat list,
 v2:acc:pnat list=>
      y:pnat list#
         y=_668 of rev(v1)of acc in pnat list,
  x:pnat list
]
==>acc:pnat list=>
```

```
     y:pnat list#
       y=_668 of @wave_front@(app(@wave_var@(rev(v1)),
                                  v0::nil))
              of acc in pnat list

Function Substitution:

hou_subst(lambda(q454,lambda(q453,app(q454,q453))),q452)
                  /* q452 is the ground label for _668

H==>NewG:
[v0:pnat,
 v1:pnat list,
 v2:acc:pnat list=>
      y:pnat list#
          y=lambda(q454,lambda(q453,app(q454,q453)))
                   of rev(v1)of acc in pnat list,
  x:pnat list
]
==>acc:pnat list=>
     y:pnat list#
       y=app(rev(v1),@wave_front@(v0:: @wave_var@(acc)))
                                              in pnat list

Terminating method at depth 0:

tr_gen_strat(seq(x:pnat list=>
                   acc:pnat list=>
                     y:pnat list#
                       y=app(rev(x),acc) in pnat list,
                 new[gen])
             then [induction(v0::v1,x:pnat list)
```

```
then [ apply(sym_eval([..]))
       then repeat intro
       then intro(acc)
       then apply(tautology(...))
     ,
       wave([2,1,2,1,2,2],
            [h:pnat=>
              l:pnat list=>
               rev(h::l)=app(rev(l),
                             h::nil)
                         in pnat list,
             left
            ]
           )
       then wave([2,1,2,2],
                 [x:pnat list=>
                   y:pnat=>
                    z:pnat list=>
                     app(x,y::z)
                      =app(app(x,
                               y::nil),
                           z)
                       in pnat list,
                  right])
       then strong_fertilize(v2)
     ]
,
 beta_reduce(gen)
 then repeat_intro_and_copy(gen,NewHyp)
 then elim(NewHyp,on(nil),new[je])
 then [ wfftacs
      ,
```

```
                elim(je,new[jv,jw,jl])
                then intro(jv)
                then [wfftacs,
                      rewrite(jw),
                      wfftacs
                     ]
               ]
          ]
)
```

B.2 Factorial

```
| ?- display.
trfac: [] incomplete autotactic(idtac)
==> x:pnat=>y:pnat#y=fac(x)in pnat
by _


yes
| ?- dplan.

H==>NewG:
[v0:pnat,
 v1:acc:pnat=>y:pnat#y=_5865 of fac(v0)of acc in pnat,
  x:pnat
]
==>acc:pnat=>
     y:pnat#
       y=_5865 of @wave_front@(times(@wave_var@(fac(v0)),
                                                  s(v0)))
                                of acc in pnat

Function Substitution:

hou_subst(lambda(q289,lambda(q288,times(q289,q288))),q287)
                     /* q287 is the ground label for _5865

H==>NewG:
[v0:pnat,
 v1:acc:pnat=>
     y:pnat#
```

```
        y=lambda(q289,lambda(q288,times(q289,q288)))
                              of fac(v0)of acc in pnat,
  x:pnat
]
==>acc:pnat=>
      y:pnat#
         y=times(fac(v0),@wave_front@(times(s(v0),
                                        @wave_var@(acc)))
               ) in pnat

Terminating method at depth 0:

tr_gen_strat(seq(x:pnat=>
                  acc:pnat=>
                    y:pnat#y=fac(x)*acc in pnat,new[gen])
            then [induction(s(v0),x:pnat)
                  then [ apply(sym_eval([....]))
                         then repeat intro
                         then intro(acc)
                         then apply(tautology(...))
                        ,
                         wave([2,1,2,1,2,2],
                              [n:pnat=>
                                fac(s(n))=fac(n)*s(n)
                                              in pnat,
                               left
                              ]
                             )
                         then wave([2,1,2,2],
                                   [a:pnat=>
                                      b:pnat=>
                                        c:pnat=>
```

```
                                    a* (b*c)= a*b*c
                                            in pnat,
                              right
                             ]
                            )
                  then strong_fertilize(v1)
                ],
          beta_reduce(gen)
          then repeat_intro_and_copy(gen,NewHyp)
          then elim(NewHyp,on(s(0)),new[je])
          then [ wfftacs
                 ,
                  elim(je,new[jv,jw,jl])
                  then intro(jv)
                  then [wfftacs,
                        rewrite(jw),
                        wfftacs
                       ]
                ]
     ]
)
```

B.3 Length

```
?- display.
trlength: [] incomplete autotactic(idtac)
==> x:pnat list=>y:pnat#y=length(x)in pnat
by _


yes
| ?- dplan.

H==>NewG:
[v0:pnat,
 v1:pnat list,
 v2:acc:pnat=>y:pnat#y=_664 of length(v1)of acc in pnat,
  x:pnat list
]
==>acc:pnat=>
     y:pnat#
       y=_664 of @wave_front@(s(@wave_var@(length(v1))))
              of acc in pnat

Function Substitution:
hou_subst(lambda(q256,lambda(q255,plus(q255,q256))),q254)
                    /* q254 is the ground label for _664

H==>NewG:
[v0:pnat,
 v1:pnat list,
 v2:acc:pnat=>
     y:pnat#
```

```
       y=lambda(q256,lambda(q255,plus(q255,q256)))
                       of length(v1) of acc in pnat,
x:pnat list
]
==>acc:pnat=>
     y:pnat#
       y=plus(@wave_front@(s(@wave_var@(acc))),length(v1))
                                                  in pnat

Terminating method at depth 0:

tr_gen_strat(seq(x:pnat list=>
                     acc:pnat=>
                           y:pnat#y=length(x)+acc in pnat,
                 new[gen])
             then [ induction(v0::v1,x:pnat list)
                    then [ apply(sym_eval([..]))
                           then repeat intro
                           then intro(acc)
                           then apply(tautology(...))
                          ,
                           wave([2,1,2,1,2,2],
                                [h:pnat=>
                                  l:pnat list=>
                                   length(h::l)=
                                     s(length(l)) in pnat,
                                 left
                                ]
                               )
                           then wave([2,1,2,2],
                                     [x:pnat=>
                                        y:pnat=>
```

```
                                    s(x)+y=x+s(y)
                                          in pnat,
                                 left
                                ]
                               )
                      then strong_fertilize(v2)
                ],
          beta_reduce(gen)
          then repeat_intro_and_copy(gen,NewHyp)
          then elim(NewHyp,on(0),new[je])
          then [ wfftacs
                 ,
                 elim(je,new[jv,jw,jl])
                 then intro(jv)
                 then [wfftacs,
                       rewrite(jw),
                       wfftacs
                      ]
                ]
        ]
)
```

B.4 Summation

```
| ?- display.
trsigma0: [] incomplete autotactic(idtac)
==> x:pnat=>g:(pnat=>pnat)=>y:pnat#y=sigma0(g,x)in pnat
by _


yes
| ?- dplan.

H==>NewG:
[v0:pnat,
 v1:g:(pnat=>pnat)=>
   acc:pnat=>
     y:pnat#y=_812  of sigma0(g,v0)of acc in pnat,
x:pnat
]
==>g:(pnat=>pnat)=>
     acc:pnat=>
       y:pnat#
         y=_812 of @wave_front@(plus(
                                  @wave_var@(sigma0(g,v0)),
                                  g(s(v0)))
                                )
                of acc in pnat

Function Substitution:
hou_subst(lambda(q763,lambda(q762,plus(q763,q762))),q761)
                     /* q761 is the ground label for _812
```

```
H==>NewG:
[v0:pnat,
 v1:g:(pnat=>pnat)=>
    acc:pnat=>
       y:pnat#
          y=lambda(q763,lambda(q762,plus(q763,q762)))
                      of sigma0(g,v0)of acc in pnat,
  x:pnat
]
==>g:(pnat=>pnat)=>
     acc:pnat=>
       y:pnat#
         y= plus(sigma0(g,v0),
                 @wave_front@(plus(g(s(v0)),
                                   @wave_var@(acc))))
                                              in pnat

Terminating method at depth 0:

tr_gen_strat(seq(x:pnat=>
                   g:(pnat=>pnat)=>
                     acc:pnat=>
                       y:pnat#y=sigma0(g,x)+acc in pnat,
             new[gen])
             then [ induction(s(v0),x:pnat)
                    then[ apply(sym_eval([.]))
                          then repeat intro
                          then intro(acc)
                          then apply(tautology(...))
                         ,
                          wave([2,1,2,1,2,2],
                               [m:pnat=>
```

```
                    f:(pnat=>pnat)=>
                     sigma0(f,s(m))=
                       f(s(m))+sigma0(f,m)
                                  in pnat,
                   left
                  ]
                 )
            then wave([2,1,2,2],
                      [x:pnat=>
                        y:pnat=>
                         z:pnat=>
                          x+(y+z)=x+y+z
                                 in pnat,
                       right
                      ]
                     )
            then strong_fertilize(v1)
          ],
     beta_reduce(gen)
     then repeat_intro_and_copy(gen,NewHyp)
     then elim(NewHyp,on(0),new[je])
     then [ wfftacs
          .
           elim(je,new[jv,jw,jl])
           then intro(jv)
           then[wfftacs,
                rewrite(jw),
                wfftacs
               ]
          ]
    ]
)
```